OS/2 Application Binary Interface for PowerPC (32-bit)

Release 1

December 8, 1995 7:31 pm

The information in this document is not final and is still under development and subject to change at any time. This document is being furnished by IBM for evaluation/development feedback purposes only and IBM does not guarantee that IBM will make this document generally available.

THE INFORMATION FURNISHED HEREIN IS ON AN “AS-IS” BASIS, AND IBM MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL IBM BE LIABLE FOR ANY DAMAGES ARISING FROM THE USE OF THE INFORMATION CONTAINED HEREIN, INCLUDING INFRINGEMENT OF ANY PROPRIETARY RIGHTS, OR FOR ANY LOST PROFITS OR OTHER INCIDENTAL AND/OR CONSEQUENTIAL DAMAGES, EVEN IF IBM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY, 10577.

The following copyright notice protects this document under the Copyright laws of the United States and other countries which prohibits such actions as, but not limited to, copying, distributing, modifying, and making derivative works.

© Copyright International Business Machines Corporation, 1994-1995. All Rights Reserved.

Notice to US Government Users - Documentation related to restricted rights - Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

The following terms are trademarks of International Business Machines Corporation in the United States and/or other countries:

IBM
Operating System/2
OS/2
PowerPC
PowerPC Architecture
SOMObjects
System Object Model

1 Introduction 1
1.1 Revision Control 2

2 Instruction Set 3
2.1 Restricted Instructions 4

3 Data Representation 5
3.1 Byte Ordering 6
3.2 Scalar Types - Size and Alignment 8
3.2.1 8-bit Integer 8
3.2.2 16-bit Integer 8
3.2.3 32-bit Integer 8
3.2.4 64-bit Integer 9
3.2.5 Pointer 9
3.2.6 Single Precision Floating Point 10
3.2.7 Double Precision Floating Point 10
3.2.8 Extended Precision Floating Point 10
3.3 Aggregates and Unions - Alignment and Padding 12
3.4 Unaligned Data Access 14
3.5 Bit Fields 16
3.6 UTF-8 19

4 Procedure Linkage Conventions 21
4.1 Registers 22
4.2 Stack Frames 25
4.3 Parameter Passing 28
4.3.1 Variable Argument Lists 30
4.3.1.1 C Language Implementation 31
4.4 Return Values 34

5 System Object Model (SOM) Binary Interface 35
5.1 Addressing, Calling Conventions, and Register Usage 37
5.2 SOM Class Library Structure 38
5.3 SOM Objects and Object References 41
5.4 SOM Method Table 42
5.5 SOM Ids 43
5.6 Basic Operations 44
5.7 Method Resolution Mechanisms 45
5.7.1 Using Offset Method Resolution 45
5.7.2 Using Name Lookup Method Resolution 46
5.7.3 Using Dispatch Method Resolution 47
5.8 SOM Kernel Functions 48
5.8.1 Required Functions 48
5.8.2 Optional Functions 48
5.8.3 Obsolete Functions 48
5.9 SOM Kernel External Variables 49
5.9.1 Required External Variables 49

5.9.2 Optional External Variables 49
5.10 SOM Kernel Class' Release Order 50
5.10.1 SOMObject 50
5.10.2 SOMClassMgr 50
5.10.3 SOMClass 50

6 System Object Exception Handling 53

7 Execution Model 55
7.1 Code Model 56
7.2 Function Tags 58
7.2.1 Long Form Function Tag Information 60
7.3 Code Examples 63
7.3.1 Function Prologue and Epilogue 63
7.3.2 Static Data Access 71
7.3.3 Function Calls 73
7.3.4 Dynamic Stack Space Allocation 74

8 Resource File Format 77
8.1 Resource File 78
8.1.1 Resource File Header 78
8.1.2 Resource File Identification 79
8.1.3 Resource File PowerPC Processor-specific Information 80
8.2 Resource Collection 81
8.2.1 Resource Header 81
8.3 Resource Item 83

9 Object and Load Module File Format 85
9.1 ELF 86
9.1.1 ELF Operating System-specific Information 86
9.1.1.1 Sections 86
9.1.1.1.1 Special Sections 88
9.1.1.2 Symbol Table 89
9.1.1.2.1 Symbol Values 89
9.1.1.3 Operating System Information 91
9.1.1.3.1 OS/2-specific Information 92
9.1.1.4 Import Table 92
9.1.1.5 Export Table 94
9.1.1.6 Resource Collection 95
9.1.1.7 Segments 95
9.1.1.7.1 Segment Permissions 95
9.1.1.7.2 Segment Contents 97
9.1.1.8 Dynamic Segment 99
9.1.1.9 Initialization and Termination Functions 102
9.1.1.10 Hash Table 103
9.1.2 ELF PowerPC Processor-specific Information 104
9.1.2.1 ELF Header 104

9.1.2.1.1 Machine Identification 104
9.1.2.2 Sections 105
9.1.2.2.1 Special Sections 105
9.1.2.3 Relocation 106
9.1.2.3.1 Relocation Types 106
9.1.2.4 Dynamic Segment 111
9.2 DWARF 114
9.2.1 DWARF PowerPC Processor-specific Information 114
9.2.1.1 Register Numbers 114

10 Object Library File Format 117
10.1 Archive File Format 118
10.2 Library File Format 119
10.2.1 LIB File Layout 119
10.2.2 LIB Header 119
10.2.3 LIB Members 120
10.2.4 LIB Special Members 121
10.2.4.1 Symbol Table Member 121
10.2.4.2 Long File Name String Table Member 123
10.2.4.3 Full File Name String Table Member 123

11 Process Creation and Dynamic Loading 125
11.1 Process Virtual Address Space 126
11.2 Process Initialization 131
11.2.1 OS/2 Process 131
11.2.1.1 Dynamic Linking of Shared Services Dynamic Link Libraries 131
11.2.2 Shared Services Process 132
11.2.3 OS/2 Dynamic Link Library Initialization 132
11.3 Process Termination 133
11.3.1 OS/2 Dynamic Link Library Termination 133
11.4 Thread Information Block 134
11.5 Global Offset Table (GOT) 135
11.6 Procedure Linkage Table (PLT) 137

Appendix A Compiler Support Extensions 141
A.1 ELF 142
A.1.1 Sections 142
A.1.1.1 COMDAT Section 143
A.1.1.2 Symbol Name Demangling 144
A.1.1.3 Default Library 145
A.1.2 Note Information 145
A.1.2.1 Browser Information 146
A.1.2.1.1 Browser Information Records 146
A.1.2.2 Version Information 147
A.1.2.3 Description Information 147

1 Introduction

This document describes the system interfaces for compiled application programs that will run on operating systems built on the IBM Microkernel technology for the PowerPC Architecture.

This document includes by reference other generally available documents as necessary. The reader may find the following documents of interest:

- *The PowerPC Architecture: A Specification for A New Family of RISC Processors*, Second Edition, IBM Corporation (ISBN 1-55860-316-6)
- *PowerPC Microprocessor Family: The Programming Environments*, IBM/Motorola (IBM order number MPRPPCFPE-01 or Motorola order number MPCFPE/AD)
- *PowerPC 603 RISC Microprocessor User's Manual*, IBM/Motorola (IBM order number MPR603UMU-01 or Motorola order number MPC603UM/AD)
- *Executable and Linking Format (ELF)*, Tool Interface Standards Committee (Review Draft Version 1.1a)
- *Tool Interface Standards Portable Formats Specification*, Tool Interface Standards Committee (Version 1.0)
- *System V Application Binary Interface*, Third Edition, UNIX System Laboratories (ISBN 0-13-100439-5)
- *System V Application Binary Interface, PowerPC Processor Supplement,* Sun Microsystems (Draft dated March, 1995)

 Where possible, this ABI has maintained compatibility with *System V Application Binary Interface, PowerPC Processor Supplement*. Incompatibilities are noted in the document by "**System V ABI Note**" comments.
- *SOMObjects Developer Toolkit Programmer's Reference Manual*, IBM Corporation

1.1 Revision Control

This material was compiled from a variety of sources and was edited by BJ Hargrave, IBM Personal Software Products. Comments can by sent to *hargrave@austin.ibm.com*.

All changes from the previous draft are marked with revision bars.

1. Release 1 (December 8, 1995 7:31 pm)

 This is the first Release of this document.

2 Instruction Set

The PowerPC Architecture is defined in *PowerPC Architecture*, IBM Corporation. *PowerPC Architecture* describes both the 32-bit and 64-bit portions of the architecture. This ABI document only defines a 32-bit ABI for PowerPC-based operating systems. Also see *PowerPC Microprocessor Family: The Programming Environments*, IBM/Motorola, for details on a 32-bit implementation of the PowerPC Architecture.

ABI conforming programs containing machine instructions must use the 32-bit PowerPC instruction set including the instruction encodings and semantics as defined by the architecture.

A processor must implement the instruction set of the architecture, perform the operations indicated by the instructions and produce the expected results. No performance constraints are levied by the ABI. A software emulation of the processor architecture could be ABI conforming.

Note: The use of PowerPC 601 instructions which are not part of the PowerPC Architecture is not ABI conforming. Programs using these instructions will function on the PowerPC 601 but will not on other PowerPC implementations.

2.1 Restricted Instructions

An ABI conforming processor may implement the optional instructions from the PowerPC Architecture as well as instructions not in the architecture. However, programs that use these instructions will not be ABI conforming.

All instructions that are neither privileged nor optional can be assumed to exist and function properly and may be used by ABI conforming programs. However, the following instructions that handle non-scalar data are *not* ABI conforming and may not be used by ABI conforming programs.

Table 2-1: Load/Store String Instructions

Mnemonic	Description
`lswi`	load string word immediate
`lswx`	load string word indexed
`stswi`	store string word immediate
`stswx`	store string word indexed

Table 2-2: Load/Store Multiple Instructions

Mnemonic	Description
`lmw`	load multiple word
`stmw`	store multiple word

These instructions all generate alignment exceptions when executed in the Little Endian mode of the processor.

3 Data Representation

The PowerPC Architecture supports the following operand sizes:

Table 3-1: Operand types and sizes

Operand	Length	$Addr_{28:31}$ if aligned
Byte	8 bits	*bbbb*
Halfword	16 bits	*bbb*0
Word	32 bits	*bb*00
Doubleword	64 bits	*b*000
Quadword	128 bits	0000

Note: An “*b*” indicates that the bit position can contain either 0 or 1.

Note: Although not permitted as storage operands by the PowerPC Architecture, quadwords are shown to demonstrate alignment and size.

3.1 Byte Ordering

Byte ordering defines how the bytes that make up the larger, multi-byte operands are ordered in memory. Big Endian ordering means that the most significant byte (msb) is located in the lowest addressed byte position in the operand (byte 0). Little Endian ordering means that the least significant byte (lsb) is located in the lowest addressed byte position in the operand (byte 0).

The PowerPC Architecture supports both Big Endian or Little Endian byte ordering. This document defines an ABI based upon the Little Endian byte ordering.

The following figures illustrate the bit and byte numbering within the various size operands. Little Endian byte numbers are in the upper right and Big Endian byte numbers are in the upper left. Bit numbers appear in the lower corners.

Figure 3-1: Halfword

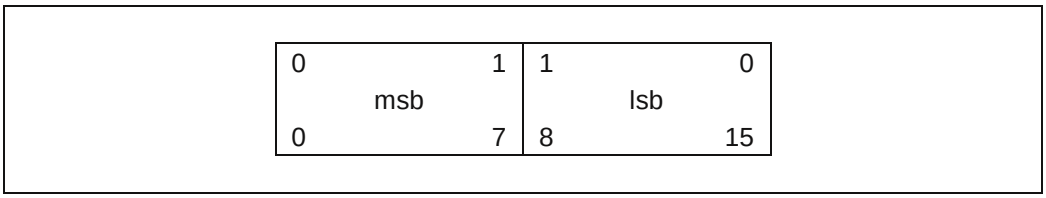


Figure 3-2: Word

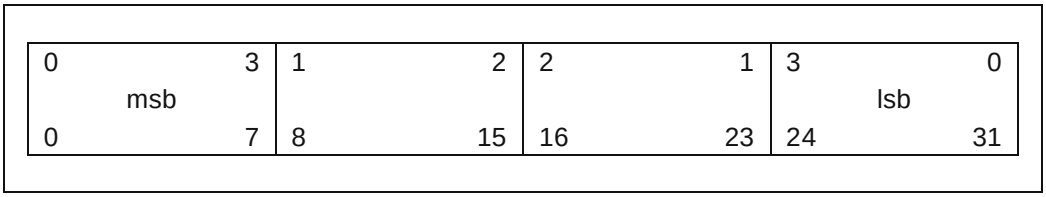


Figure 3-3: Doubleword

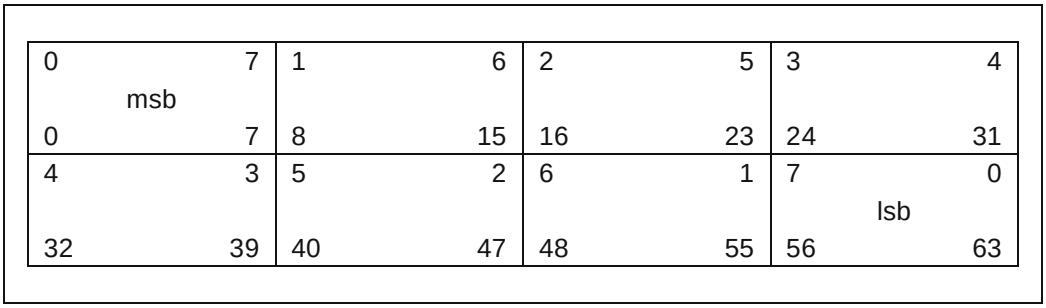
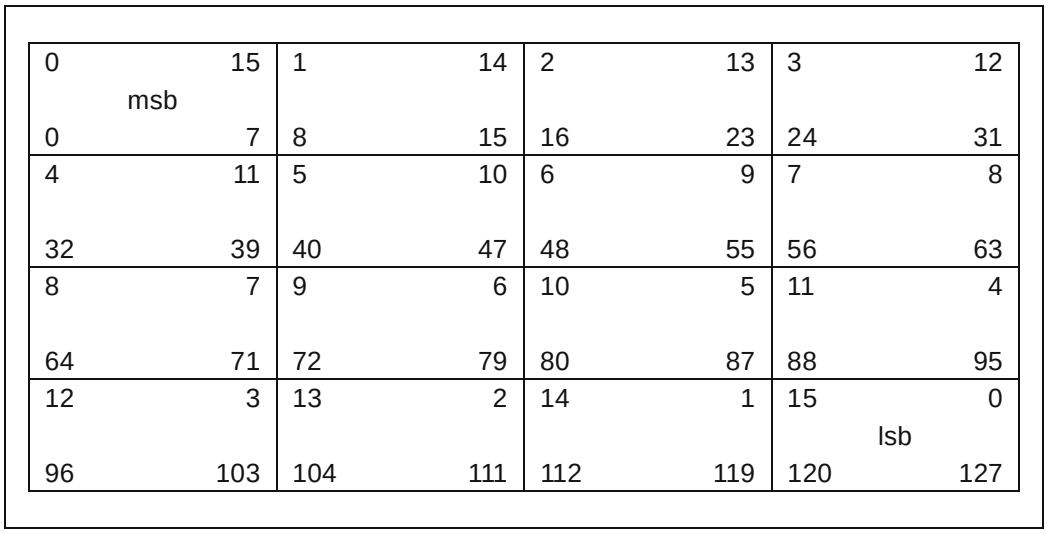


Figure 3-4: Quadword



3.2 Scalar Types - Size and Alignment

This section describes the mapping of C/C++ scalar data types onto the PowerPC Architecture.The **Size** column indicates the size of the data type in bytes. The **Alignment** column indicates the preferred alignment for the data type. If the data type is not aligned to the preferred alignment then alignment exceptions may occur when accessing the data from memory. Scalar data types on the PowerPC are aligned on their “natural” boundaries. That is, their preferred alignment is equal to their size.

3.2.1 8-bit Integer

Table 3-2: 8-bit Integer

Operand	C/C++ type	Size	Alignment
signed byte	`signed char`	1	1
unsigned byte	`char` `unsigned char`	1	1

3.2.2 16-bit Integer

Table 3-3: 16-bit Integer

Operand	C/C++ type	Size	Alignment
signed halfword	`short` `signed short`	2	2
unsigned halfword	`unsigned short` `wchar_t` `UniChar`	2	2

Note: `UniChar` is the data type representing Unicode characters.

3.2.3 32-bit Integer

Table 3-4: 32-bit Integer

Operand	C/C++ type	Size	Alignment
signed word	`int` `signed int` `long` `signed long`	4	4
unsigned word	`unsigned int` `unsigned long`	4	4

Note: The type of an enumeration data type, *e.g.* the C/C++ type `enum`, is the smallest integral type that can contain all of the enumeration values.

3.2.4 64-bit Integer

Table 3-5: 64-bit Integer

Operand	C/C++ type	Size	Alignment
signed doubleword	`__int64` `long long` `signed long long`	8	8
unsigned doubleword	`__uint64` `unsigned long long`	8	8

Note: Support for 64-bit integers is currently implemented by some 32-bit compilers as `long long` although this data type is not part of the current ANSI C specification.

System V ABI Note: The *System V Application Binary Interface, PowerPC Processor Supplement* defines 64-bit integers (`long long`) and their semantics, but specifically indicates that their use is non-conforming. 64-bit integers are part of this ABI and their use is conforming. The definition and semantics are the same in both ABIs.

3.2.5 Pointer

Table 3-6: Pointer

Operand	C/C++ type	Size	Alignment
unsigned word	`any-type *` `any-type (*) ()`	4	4

Note: This is a 32-bit ABI, therefore pointers are 32-bits in size.

3.2.6 Single Precision Floating Point

Table 3-7: Single Precision Floating Point

Operand	C/C++ type	Size	Alignment
word, single precision (IEEE 754)	`float`	4	4

3.2.7 Double Precision Floating Point

Table 3-8: Double Precision Floating Point

Operand	C/C++ type	Size	Alignment
doubleword, double precision (IEEE 754)	`double`	8	8

3.2.8 Extended Precision Floating Point

Table 3-9: Extended Precision Floating Point

Operand	C/C++ type	Size	Alignment
quadword, extended precision (ordered pair of double precision floating point values)	`long double`	16	16 (for structures, only 8 byte alignment is required otherwise)

The extended precision floating point format is an ordered pair of double precision values. Together they represent the number which is their algebraic sum. The "higher-order" double precision value is larger in magnitude and is stored in the lower-addressed doubleword of the quadword. The "lower-order" double precision value is smaller in magnitude and is stored in the higher-addressed doubleword of the quadword. The two double precision values are defined and manipulated such that the lower-order double precision value “extends” the precision of the higher-order double precision value.

- The lower-order value typically has an exponent which is 53 less than that of the higher-order value, but this is not mandated.
- The signs of the two double precision values may differ.
- The exponent range is no greater than double precision.
- As the absolute value of the extended precision quantity becomes very small, the additional precision provided by the lower-order double precision value decreases, until for denormalized numbers, the precision is the same as double precision.

The extended precision floating point format need not be conformant with IEEE 754. The following deviations are permitted.

- The only rounding mode that must be supported in round-to-nearest.
- The IEEE special numbers NaN and INF may not be fully supported. Even basic operations such as addition and subtraction may not propagate NaN and INF correctly.
- The IEEE status flags in `FPSCR` for overflow, underflow, etc. may not be correctly set, even by basic operations such as addition and subtraction.

System V ABI Note: The *System V Application Binary Interface, PowerPC Processor Supplement* defines extended precision floating point (`long double`) differently. It is defined there as a 128-bit IEEE 754 conforming data type with a sign bit, a 15 bit exponent with a bias of -16383, and a 112 bit fraction with a leading implicit bit. It is treated as a structure for the purposes of parameter passing and return values.

3.3 Aggregates and Unions - Alignment and Padding

Aggregates (structures and arrays) and unions are aligned using the alignment of the most strictly aligned component, *i.e.* the component with the largest alignment dictates the alignment of the aggregate or union in which it is contained. Each component is assigned the lowest available offset with the appropriate alignment. This may require internal padding, depending on the size of the previous component. The size of on aggregate or union is always a multiple of its alignment. Thus a structure or union may require “tail” padding to meet size and alignment constraints.

The following figures illustrate structure and union member alignment and packing for Little Endian byte ordering. Little Endian byte numbers are in the upper right.

Figure 3-5: Small Structure

```
struct
{
   char c;
};
```

Byte aligned, sizeof is 1.

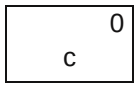


Figure 3-6: Structure with No Padding

```
struct
{
   char c;
   char d;
   short s;
   long n;
};
```

Word aligned, sizeof is 8.

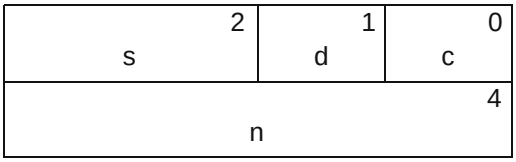


Figure 3-7: Structure with Padding

```
struct
{
   char c;
   double d;
   short s;
};
```

Doubleword aligned, sizeof is 24. The structure is padded both internally, to maintain proper alignment of the members, and at the “tail” to maintain proper size of the structure.

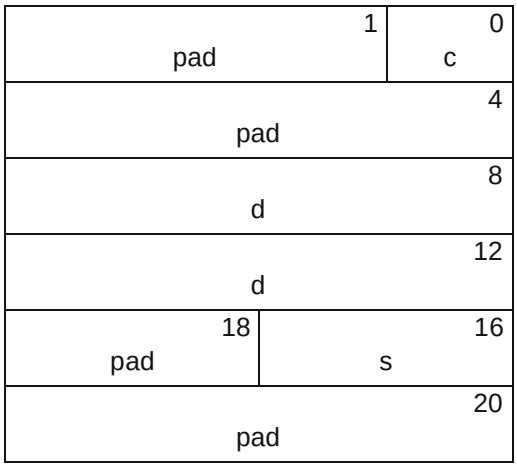


Figure 3-8: Structure with Packing on 1 Byte Boundary

```
struct
{
   char c;
   double d;
   short s;
   long n;
};
```

Byte aligned, sizeof is 15. The structure is not padded either internally or at the “tail”. Both the structure itself and its members are improperly aligned for direct access.

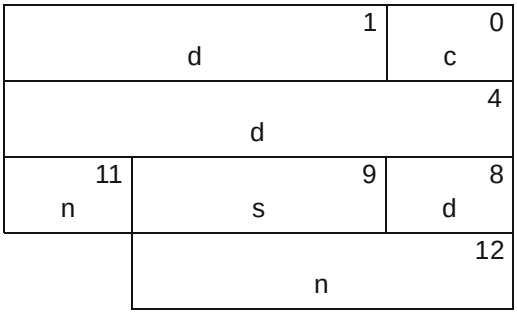
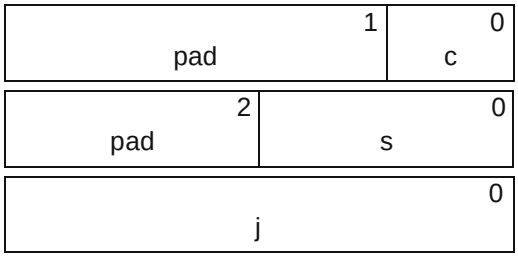


Figure 3-9: Union Allocation

```
union
{
   char c;
   short s;
   int j;
};
```

Word aligned, sizeof is 4.



3.4 Unaligned Data Access

Since PowerPC processors do not support unaligned data access in Little Endian mode without causing an alignment exception, compiler assistance is necessary to support existing code with poorly aligned data structures. Many legacy applications are from the Intel x86 which is very forgiving of unaligned data. Unaligned data accesses on the Intel x86 chips are penalized only cycles rather than the expensive alignment exceptions of the Little Endian mode of the PowerPC.

One of the key areas of legacy applications is OS/2, an operating system that originated on the Intel x86 architecture. In fact there are numerous system data structures that are not “naturally” aligned. For this reason, compiler support is critical to facilitate the porting of applications to OS/2 for the PowerPC and the sharing of persistent data between both platforms. While this compiler support is not required to produce ABI-compliant code, it is highly recommended. In short, the compiler support involves using multiple instructions to access an unaligned scalar quantity without causing an alignment exception. For example, in Figure 3-8, “Structure with Packing on 1 Byte Boundary”, on page 13, member `n` in the structure is improperly aligned. Assuming that `r3` points to the structure and the structure is aligned on a word boundary, the following code example would load the value of `n` into `r4` without causing an alignment exception.

```
lwz   %r4,12(%r3) ;load bytes 1,2,3 into r4
lbz   %r5,11(%r3) ;load byte 4 into r5
slwi  %r4,%r4,8   ;shift bytes 1,2,3 left one byte
or    %r4,%r4,%r5 ;or all bytes together in r4
```

Similar techniques can be used to access and update scalars with other misalignments. All multi-byte scalar data types listed in § 3.2, “Scalar Types - Size and Alignment”, should be handled. The specific techniques are left to the compiler vendor.

The compiler can ensure that all automatic and global data are “naturally” aligned. The compiler can then determine, at compile time, the specific alignment problems of individual members of an aggregate and generate the appropriate code to avoid alignment exceptions when these members are accessed.

When dealing with pointers, however, the alignment of the data referenced by the pointer is unknown at compile time. Therefore one of the following choices must be made.

1. Code is generated to assemble the data item by accessing only individual bytes. This choice makes no assumptions about the alignment of the data.
2. Code is generated to determine at runtime if the data item being accessed is “naturally” aligned. If the data item is aligned it can be directly accessed. Otherwise it accessed by individual bytes
3. Code is generated accessing the data item assuming it is naturally aligned. If the data item is unaligned, an alignment exception will be generated and the alignment exception handler will be invoked. The exception handler will then complete the instruction, accessing the data item by individual bytes, and return control to the next instruction. This choice assumes that most pointers will point to naturally aligned data items.

It is recommended that compiler support for unaligned data access be activated by either

1. The use of a structure packing pragma which causes a structure to have members which are not “naturally” aligned or to have the structure’s size not be a multiple of the “natural” alignment of the structure.
2. The use of a *new* pragma that signifies that a structure may contain unaligned members or a pointer variable may point to unaligned data.

3.5 Bit Fields

Structure and unions may have bit fields which define integral objects with a specified number of bits.

Table 3-10: Bit Field Ranges

Bit field type	Width (w)	Range
`signed char` `char` `unsigned char`	1 to 8 bits	-2^{w-1} to $2^{w-1}-1$ 0 to $2^{w}-1$ 0 to $2^{w}-1$
`signed short` `short` `unsigned short`	1 to 16 bits	-2^{w-1} to $2^{w-1}-1$ 0 to $2^{w}-1$ 0 to $2^{w}-1$
`signed int` `int` `unsigned int` `enum`	1 to 32 bits	-2^{w-1} to $2^{w-1}-1$ 0 to $2^{w}-1$ 0 to $2^{w}-1$ 0 to $2^{w}-1$
`signed long` `long` `unsigned long`	1 to 32 bits	-2^{w-1} to $2^{w-1}-1$ 0 to $2^{w}-1$ 0 to $2^{w}-1$

Bit fields that are not specified as either signed or unsigned always have non-negative values. Bit fields are subject to the same rules of size and alignment as their declared type in addition to the following.

- Bit fields are allocated within a storage unit from least significant bit to most significant bit.
- Bit fields do not cross the boundaries of its declared type. That is, a bit field cannot span between multiple storage units of its declared type.
- Bit fields must share a storage unit with other members (either bit field or non-bit field), if and only if there is sufficient space within the storage unit of its declared type.
- The declared type of an unnamed bit field does not affect the alignment of a structure or union. However, they do cause alignment from the beginning of the structure based upon their declared type. An unnamed bit field of zero-width prevents any further member (either bit field or non-bit field) from residing in the storage unit of the declared type of the zero-width bit field.

The following figures illustrate structure and union member offsets for Little Endian byte ordering. Little Endian byte numbers are in the upper right. Bit numbers appear in the lower corners.

Figure 3-10: Bit Field Right-to-Left Allocation

```
struct
{
   int   j:5;
   int   k:6;
   int   m:7;
};
```

Word aligned, sizeof is 4. The structure is “tail” padded so that the size of the structure is a multiple of its alignment.

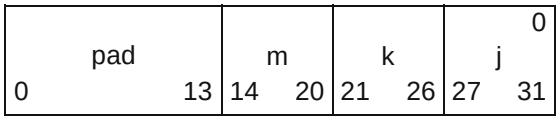


Figure 3-11: Bit Field Boundary Alignment

```
struct
{
   short s:9;
   int   j:9;
   char  c;
   short t:9;
   short u:9;
   char  d;
};
```

Word aligned, sizeof is 12.

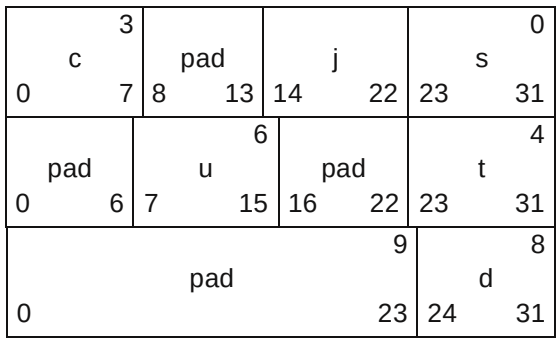


Figure 3-12: Bit Field Storage Unit Sharing

```
struct
{
   char  c;
   short s:8;
};
```

Halfword aligned, sizeof is 2.

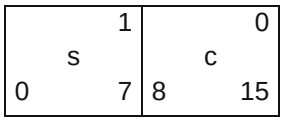
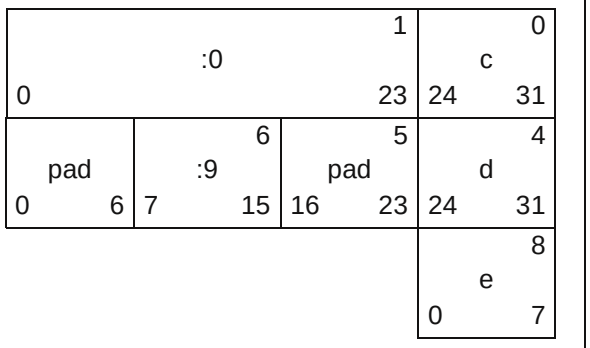


Figure 3-13: Unnamed Bit Fields

```
struct
{
   char  c;
   int   :0
   char  d;
   short :9;
   char  e;
};
```

Byte aligned, sizeof is 9. The declared types of the unnamed bit fields do not affect the alignment of the structure.



As can be seen in the above figures, bit fields of the largest integral types (`int` or `long`) pack more densely than bit fields of smaller types. The smaller types can be used to force a particular alignment.

3.6 UTF-8

UTF-8 is a transformation of Unicode characters such that there are no embedded null bytes in a character string. This allows Unicode character strings to be encoded in a way that they may be used in places that historically only support single-byte character strings which may use the null-byte as a string terminator. UTF-8 encoded character strings are specified in several places in this ABI.

The UTF-8 transformation encodes Unicode characters in the range `0 – 0xFFFF` using multi-byte characters of 1, 2 and 3 bytes in length. Single-byte characters are reserved for the ASCII characters in the range `0 – 0x7F`. These characters all have the most significant bit set to zero. For all characters which are encoded using more than one byte, the number of bytes used is indicated by the number of most significant bits which are set to one. Subsequent bytes of the multi-byte character are all of the form `10`*`xxxxxx`*.

Table 3-11: UTF-8 Transformation

Bytes	Bits	Range	Byte sequence in binary
1	7	0x0000 – 0x007F	0*vvvvvvv*
2	11	0x0080 – 0x07FF	110*vvvvv* 10*vvvvvv*
3	16	0x0800 – 0xFFFF	1110*vvvv* 10*vvvvvv* 10*vvvvvv*

The Unicode character value is just the concatenation of the *v* bits in the multi-byte encoding. When there are multiple ways to encode a value, only the shortest encoding is legal.

4 Procedure Linkage Conventions

This chapter defines the standard system function calling convention. It is applicable to both procedural and object-oriented function calls. All system interfaces require this calling convention.

Note: The standard system function calling conventions apply only to global functions. Local functions that are not reachable from other compilation units may use other calling conventions.

4.1 Registers

The PowerPC Architecture provides 32 General Purpose Registers (GPRs). Each GPR is a word (32-bit) in size and is used for integer and address computations. There are also 32 Floating Point Registers (FPRs). Each FPR is a doubleword (64-bit) in size and is used for floating point computations. There are also other registers with specific purposes. All registers (GPR, FPR and other) are global to all functions in a thread of execution.

This table gives a description of the usage of the PowerPC registers in this ABI.

Table 4-1: Register Usage

Register	Status	Usage
`r0`	Volatile†	Language specific purpose.
`r1`	Dedicated	Stack pointer. Always valid.
`r2`	Dedicated	Reserved for system use. This register points to the Thread Information Block and should not be modified by application code. See § 11.4, “Thread Information Block”, on page 134, for details.
`r3` `r4`	Volatile	Parameter passing and return values.
`r5` `r6` `r7` `r8` `r9` `r10`	Volatile	Parameter passing.
`r11` `r12` `r13`	Volatile†	Language specific purpose. **System V ABI Note:** The *System V Application Binary Interface, PowerPC Processor Supplement* defines `r13` to be a Small Data Area Pointer.

Table 4-1: Register Usage (Continued)

Register	Status	Usage
r14 r15 r16 r17 r18 r19 r20 r21 r22 r23 r24 r25 r26 r27 r28 r29 r30 r31	Non-volatile	Local variables. **Note:** The stack frame layout assumes that a contiguous set of GPRs will be allocated from r31 down towards r13. **Note:** Programming languages that require a static scope pointer (*e.g.* Pascal) shall use r31 for that purpose.
f0	Volatile	Language specific purpose.
f1 f2 f3 f4	Volatile	Parameter passing and return values.
f5 f6 f7 f8	Volatile	Parameter passing.
f9 f10 f11 f12 f13	Volatile	Language specific purpose.

Table 4-1: Register Usage (Continued)

Register	Status	Usage
f14 f15 f16 f17 f18 f19 f20 f21 f22 f23 f24 f25 f26 f27 f28 f29 f30 f31	Non-volatile	Local variables. **Note:** The stack frame layout assumes that a contiguous set of FPRs will be allocated from f31 down towards f13.
CR0 CR1 CR5 CR6 CR7	Volatile	Condition Register fields. Each is 4 bits wide. CR bit 6 (CR1, floating point invalid operation exception) is set by the caller of a variable argument list function. See § 4.3.1, “Variable Argument Lists”, on page 30 for more information.
CR2 CR3 CR4	Non-volatile	Condition Register fields. Each is 4 bits wide. Together CR0 - CR7 make up the 32 bit CR register.
LR	Volatile	Link Register. Used to hold the address to which a called function returns.
CTR	Volatile†	Count Register
XER	Volatile	Fixed Point Exception Register
FPSCR	See Usage	$FPSCR_{0:23}$ is volatile and $FPSCR_{24:31}$ is global. $FPSCR_{24:31}$ (the exception enable and rounding control bits) can be modified by routines that have such documented behavior.

† The values in these volatile registers may be modified during the transfer of control from one function to another by system “glue” code.

There are no special restrictions on the use of registers in system exception handling or signal routines.

4.2 Stack Frames

In addition to the registers, each function can have a stack frame on the current thread's stack. Register `r1` is the stack pointer and points to the current stack frame. A function must acquire a stack frame if the function calls another function or requires any of the optional stack frame components.

Figure 4-1: Stack Frame Layout

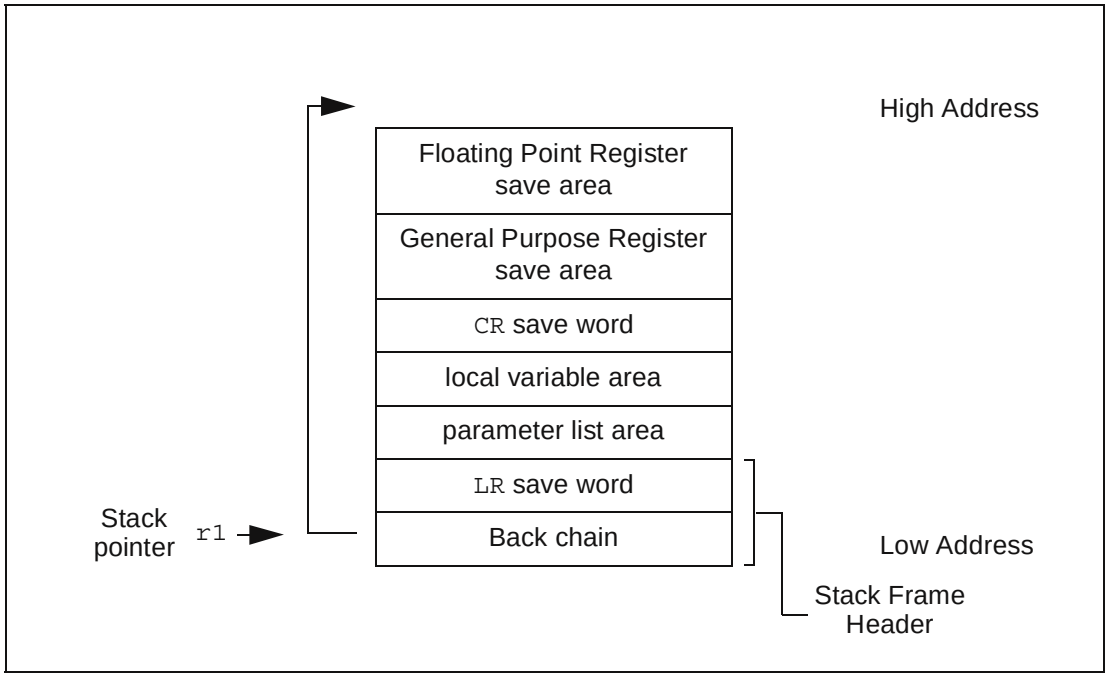


Table 4-2: Stack Frame Components

Component	Description
Stack pointer, `r1`	• The stack pointer shall always maintain 16 byte alignment, shall always point to the first word of the lowest allocated, valid stack frame and shall grow down towards low addresses. Negative references from the stack pointer are not permitted. • If a function requires a stack frame, the function's prologue shall atomically decrement the stack pointer and set the back chain pointer with one of the "Store Word with Update" instructions. Thus, the stack pointer always points to a linked list of stack frames. • When releasing a stack frame, a function's epilogue shall atomically update the stack pointer prior to returning to the calling function by either setting it to the value of the back chain pointer or incrementing it by the same amount it was decremented in the function prologue.

Table 4-2: Stack Frame Components (Continued)

Component	Description
Back chain pointer	• The back chain pointer shall always point to the base of the previously allocated stack frame. Except for the first stack frame in the stack which has a back chain pointer of 0.
`LR` save word	• When a called function creates its stack frame, it shall save the value of `LR` when the function was entered into the `LR` save word of the *callers* stack frame. • When returning from a function, all non-volatile registers shall be restored prior to restoring `LR`.
parameter list area (optional)	• The parameter list area is a variable size area allocated by the caller. • It shall be large enough to contain the arguments the caller stores in it. • It is volatile and can be modified by the called function.
local variable area (optional if stack frame padding is not required)	• There is no restriction on the use and size of this stack frame component. • Any padding of the stack frame that is necessary to maintain 16 byte alignment shall be in this stack frame component.
`CR` save word (optional)	• If a function modifies any of the non-volatile fields in `CR`, the non-volatile fields of `CR`, as they were at function entry, shall be saved.
GPR save area (optional)	• The size of the GPR save area shall be large enough to hold all the non-volatile GPRs from the lowest numbered non-volatile GPR used by the function to `r31` inclusive. • Before a function modifies the value in a non-volatile GPR, `r`*n*, the value of `r`*n*, as it was at function entry, shall be saved in the word at offset 4*(*n*-31) from the address of the highest-addressed word of the GPR save area. Thus `r31` is stored in the highest-addressed word, `r30` in the next lower-addressed word, *etc*. **Note:** `r31` may not participate in non-volatile GPR saving. See the description of the `gpr31_nosave` bit in Table 7-1, “Function Tag Word,” on page 58.

Table 4-2: Stack Frame Components (Continued)

Component	Description
FPR save area (optional)	• The size of the FPR save area shall be large enough to hold all the non-volatile FPRs from the lowest numbered non-volatile FPR used by the function to `f31` inclusive. • Before a function modifies the value in a non-volatile FPR, `fn`, the value of `fn`, as it was at function entry, shall be saved in the doubleword at offset 8*(*n*-31) from the address of the highest-addressed doubleword of the FPR save area. Thus `f31` is stored in the highest-addressed doubleword, `f30` in the next lower-addressed doubleword, *etc*.

Except for the stack frame header (the back chain pointer and the `LR` save word) and any necessary padding to maintain the 16 byte alignment of the stack frame, no space need be allocated for stack frame components which are not used. If a function does not call any functions (a “leaf” function) and does not require any of the optional stack frame components, it need not establish its own stack frame. Any padding of the stack frame that is necessary to maintain 16 byte alignment shall be wholly within the local variable area. The parameter list area shall immediately follow the stack frame header. The register save areas (`CR`, `GPR`, `FPR`) shall contain no padding.

See § 7.2, “Function Tags”, on page 58 for information on function tags which describe the usage of the register save areas (`LR`, `CR`, `GPR`, `FPR`) of the stack frame. This information can be used by exception handlers to determine the contents of the non-volatile registers as they were when a function was entered.

See § 7.3.1, “Function Prologue and Epilogue”, on page 63 for example code which demonstrates acquiring and releasing stack frames.

See § 7.3.4, “Dynamic Stack Space Allocation”, on page 74 for information on dynamically increasing the stack frame size once inside a function.

4.3 Parameter Passing

For the PowerPC Architecture, it is more efficient to pass parameters in registers (GPRs and FPRs) rather than build parameter lists in memory (or on the stack). Since all computations are performed on registers and there are a large number of registers available, it makes sense to pass parameters in registers where they can be used immediately by the called function. The number of parameters that can be directly passed in registers is limited by the register usage definition in the ABI.

Up to eight integer parameters, loaded sequentially in `r3` through `r10`, and eight floating point parameters, loaded sequentially in `f1` through `f8`, can be passed directly in registers. Only those parameter passing registers actually needed in a function call will be loaded with parameter values. The remaining unused parameter passing registers will contain undefined values.

Only when a parameter cannot be passed in a register (more than eight integer or more than eight floating point parameters or the parameter is too large to be passed in a register) must the parameter list area component of the stack frame be allocated. The parameter list area need only be allocated large enough to contain only those parameters not being passed by register. The parameter list area is partitioned into parameter words with the first parameter word (0) being at the low address of the parameter list area immediately adjacent to the stack frame header.

The following algorithm details how parameters are to be passed. For this algorithm, parameters are ordered from left (first parameter) to right and the evaluation order is unspecified. Let `gr` and `fr` be the number of the next available GPR and FPR, respectively and `param` be the address of the next available parameter word. Initialize `gr` to 3, `fr` to 1 and `param` to the address of lowest-addressed word in the parameter list area. For each parameter, proceeding left to right, select its type from the following table and take the indicated action.

Table 4-3: Parameter Passing Types and Actions

Type	Action
• **Single precision floating point** • **Double precision floating point**	If `fr` ≤ 8, load the parameter value into FPR `fr`, set `fr` to `fr`+1. Otherwise (no remaining parameter passing FPRs) treat as type **Other**.
• **Extended precision floating point**	If `fr` ≤ 7, load the lower-addressed doubleword of the parameter value into FPR `fr` and the higher-addressed doubleword of the parameter value into FPR `fr`+1. Set `fr` to `fr`+2. Otherwise (insufficient parameter passing FPRs) treat as type **Other**.

Table 4-3: Parameter Passing Types and Actions (Continued)

Type	Action
• **8-bit, 16-bit or 32-bit integer** • **Pointer** • **Structure or union** These shall be treated as a pointer to the object or to a copy of the object where necessary to enforce call-by-value semantics. The caller can pass a pointer to the object only if the callee treats the object as “constant”.	If `gr` ≤ 10, load the parameter value into GPR `gr`, set `gr` to `gr`+1. Parameter values smaller than 32 bits in size are sign or zero extended as appropriate to 32 bits. Otherwise (no remaining parameter passing GPRs) treat as type **Other**.
• **64-bit integer**	If `gr` < 9 and even, set `gr` to `gr`+1. If `gr` ≤ 9, load the lower-addressed word of the parameter value into GPR `gr` and the higher-addressed word of the parameter value into GPR `gr`+1. Set `gr` to `gr`+2. Otherwise (insufficient parameter passing GPRs) treat as type **Other**.
• **Other** Parameters not handled above are passed in the parameter list area of the callers stack frame.	8-bit, 16-bit and 32-bit integer (sign or zero extended as appropriate to 32 bits) and pointer (including implicit pointers to structure or union) parameters are considered to have word size and alignment. Single precision floating point (converted to double precision floating point representation), double precision floating point and 64-bit integer parameters are considered to have doubleword size and alignment. Extended precision floating point parameters are considered to have quadword size and doubleword alignment. Round `param` up to a multiple of the alignment required of the parameter and copy the parameter value byte-wise starting with the lowest addressed byte into bytes `param`[0] through `param`[size-1]. Set `param` to `param`+size.

The values in registers and the bytes in the parameter list area which were skipped over by the actions above are undefined.

Note: When passing single precision floating point values, either in an FPR or as a doubleword in the parameter list area, the value shall be rounded to single precision, if not already rounded to single precision, before being passed.

Figure 4-2: Parameter Passing Example

```
typedef struct
{
   int a, b;
   double dd;
} sparm;
sparm s, t;
int c, d, e, f, g, h;
long double ld;
double ff, gg, hh, ii, jj, kk, ll, mm, nn;

func(c, ff, d, gg, e, hh, f, ii, g, jj, h, ld, kk, ll, s, mm, t,
nn);
```

General Purpose Registers	Floating Point Registers	Parameter List Area offsets
r3: c	f1: ff	00: ll (lo)
r4: d	f2: gg	04: ll (hi)
r5: e	f3: hh	08: mm (lo)
r6: f	f4: ii	0C: mm (hi)
r7: g	f5: jj	10: nn (lo)
r8: h	f6: ld (lo)	14: nn (hi)
r9: ptr to s	f7: ld (hi)	
r10: ptr to t	f8: kk	

Note: (lo) and (hi) denote the lower-addressed and higher-addressed half of the parameter as stored in memory. The “ptr to” arguments are pointers to copies, if necessary, to preserve call-by-value semantics.

4.3.1 Variable Argument Lists

Functions should make no assumptions about the location of parameters that were passed to the function. Some otherwise portable functions assume that all arguments are passed on the stack and appear in increasing order on the stack. Functions with these assumptions have never been portable but have worked on many implementations. However, these functions will not work under this ABI as the majority of parameters are passed via register. Portable C and C++ programs should use `<stdarg.h>` to access parameters in variable argument lists.

A caller of a function that takes a variable argument list (or an unprototyped function) shall set bit 6 of `CR` if it passes one or more parameters in floating point registers. It is strongly recommended that the caller clear the bit otherwise. Bit 6 of `CR` can be set with “`creqv 6,6,6`” and cleared with “`crxor 6,6,6`”.

The notification of parameters in FPRs by setting bit 6 in `CR` can be used by variable argument list functions in two ways. First, the variable argument list function can determine if it must store the parameter passing FPRs in memory. Second, by knowing that there are no floating point parameters, the function can avoid acquiring a “floating point state” for the

process. Having no “floating point state” means that the floating point registers need not be saved and restored on context switches.

The cost of this notification is one additional non-memory reference instruction in all callers of variable argument list functions. ANSI C requires that all variable argument list functions be prototyped with a trailing ellipsis (“…”), but compiler vendors are expected to provide support for variable argument list functions in non-ANSI programs or to treat all non-prototyped functions as potentially having variable arguments.

4.3.1.1 C Language Implementation

The `va_list` type defined in `<stdarg.h>` shall have a storage representation as described below in Figure 4-3, “va_list Description”. The source representation of the `va_list` type is not specified.

Figure 4-3: `va_list` Description

```
struct __va_list
{
   char gpr;
   char fpr;
   char reserved[2];
   char *input_arg_area;
   char *reg_save_area;
};
```

Member	Description
`gpr`	Index into the array of 8 parameter passing GPRs, `r3` through `r10`, stored in the register save area. Index 0 corresponds to `r3`, 1 to `r4`, *etc*. This is initialized by `va_start` to contain the index of the first parameter passing GPR unused by parameters to the left of the ellipsis (“…”). If all parameter passing GPRs are used by parameters to the left of the ellipsis, then `gpr` shall be set to 8.
`fpr`	Index into the array of 8 parameter passing FPRs, `f1` through `f8`, stored in the register save area. Index 0 corresponds to `f1`, 1 to `f2`, *etc*. This is initialized by `va_start` to contain the index of the first parameter passing FPR unused by parameters to the left of the ellipsis (“…”). If all parameter passing FPRs are used by parameters to the left of the ellipsis, then `fpr` shall be set to 8.
`reserved`	Padding.
`input_arg_area`	Location in the parameter list area of the stack frame which may have the next variable argument passed in memory. This is initialized by `va_start` to contain the address of the first word in the parameter list area unused by parameters to the left of the ellipsis (“…”).

Figure 4-3: `va_list` Description (Continued)

`reg_save_area`	Pointer to the register save area. This pointer is initialized by `va_start` to contain the address of the register save area. The register save area shall be doubleword aligned and large enough to store the parameter passing registers, `r3` through `r10` and `f1` through `f8`. The parameter passing registers are copied to the register save area during the function prologue. The registers are stored in the register save area beginning with `r3` through `r10` optionally followed by `f1` through `f8` if `CR` bit 6 is set.

The following suggests a possible implementation for variable argument list support but it is not included in the ABI specification.

The `va_start` macro will require inline support from the compiler in order to properly initialize a `va_list` variable. The `va_arg` macro will require C runtime library support in order to manipulate the `va_list` variable. The runtime library can provide a `__va_arg` function to be used by the `va_arg` macro. The `__va_arg` function would return a pointer to the next argument of the specified type and would be defined as follows.

```
void * __va_arg ( va_list *argp, _va_arg_type type );
```

Compiler support would be necessary to convert the type specified to the `va_arg` macro to one of the values of the `_var_arg_type` type.

Figure 4-4: `_va_arg_type` Description

```
typedef enum
{
   arg_ARGPOINTER,
   arg_WORD,
   arg_DOUBLEWORD,
   arg_REAL,
   arg_DOUBLEREAL
} _va_arg_type;
```

Value	Description
`arg_ARGPOINTER`	The argument is a word in size and is passed in a parameter passing GPR, `r3` through `r10`, or in memory. The argument is treated as a pointer to the actual argument (struct or union).
`arg_WORD`	The argument is a word in size and is passed in a parameter passing GPR, `r3` through `r10`, or in memory. The argument is an 8-bit, 16-bit or 32-bit integer or pointer (sign or zero extended as appropriate if necessary.)
`arg_DOUBLEWORD`	The argument is a doubleword in size and is passed in an odd/even pair of parameter passing GPRs, `r3` through `r10`, or in memory. The argument is a 64-bit integer.

Figure 4-4: `_va_arg_type` Description (Continued)

`arg_REAL`	The argument is a doubleword in size and is passed in a parameter passing FPR, `f1` through `f8`, or in memory. The argument is a single precision floating point (converted to double precision floating point format) or double precision floating point value.
`arg_DOUBLEREAL`	The argument is a quadword in size and is passed in a pair of parameter passing FPRs, `f1` through `f8`, or in memory. The argument is an extended precision floating point value.

4.4 Return Values

The following table details how return values are to be handled.

Table 4-4: Return Value Types and Actions

Type	Action
• **Single precision floating point** • **Double precision floating point**	Return in `f1`. The return value is rounded to single precision for the return value type of single precision floating point.
• **Extended precision floating point**	Return the lower-addressed doubleword in `f1` and the higher-addressed doubleword in `f2`.
• **8-bit, 16-bit or 32-bit integer** • **Pointer**	Return in `r3`. Return values smaller than 32 bits in size are sign or zero extended as appropriate to 32 bits.
• **64-bit integer**	Return the lower-addressed word in `r3` and the higher-addressed word in `r4`.
• **Structure or union ≤ 8 bytes in size**	Return the lower-addressed word in `r3` and the higher-addressed word in `r4` as if it were first stored in an 8 byte aligned memory area. Bits beyond the last member of the structure or union are not defined.
• **Structure or union > 8 bytes in size**	These shall be returned in a memory area allocated by the caller whose address is passed to the called function as a hidden first parameter in `r3`. This causes `gr` to be initialized to 4 rather than 3 in § 4.3, “Parameter Passing”.

5 System Object Model (SOM) Binary Interface

IBM's System Object Model (or SOM) provides a language-neutral binary interface for object-oriented (OO) class libraries. A SOM class library is a collection of one or more SOM classes whose functionality is accessible via interface descriptions expressed in OMG's (Object Management Group, Inc.) Interface Definition Language (IDL). Every SOM class is implemented with a procedure that constructs and registers the class and a set of method procedures that implement the methods defined for the class. A SOM class library may be realized as a dynamically linked library (DLL) on platforms that provide dynamic linking, or a set of object files contained in a static library on platforms that do not. Programmers typically make use of SOM class libraries through *language bindings* that map the IDL interface descriptions into a particular programming language or through SOM-aware language compilers that produce code directly to the SOM binary interface.

The SOM Kernel is a distinguished class library that contains the SOM kernel classes and implements all of the basic mechanisms and general OO capabilities used in creating and manipulating all other SOM classes and their object instances. Except for the fact that all of the SOM Kernel *functions* and *global variables* are also included as part of this library, it is in most other respects no different from any other SOM class library.

All of the OO constructs provided by SOM are ultimately expressed in terms of a small number of function calls and data structures. Some of these are used to construct classes and exercise the method resolution mechanisms (among other things) and hence, in turn, permit the instantiation of objects and the invocation of methods on objects. The remainder of the SOM Kernel is made up of methods supplied with the three built-in classes (`SOMObject`, `SOMClass`, and `SOMClassMgr`). Generally speaking, most of SOM is implemented as methods (in preference to functions), to permit user specialization through normal OO subclassing, but some critical or heavily used constructs are implemented as ordinary functions.

The binary interface to SOM is the substrate upon which the language bindings are built. It consists of

- Addressing, calling conventions, and register usage.
- Class library organization.
- The representation of objects and object references.
- The data structures used for method resolution and basic operations.
- The kernel functions used to perform class construction and method resolution.

The remaining SOM Kernel functions and methods can be invoked and applied using this basic set of mechanisms.

Some of the functions included in IBM implementations of the SOM Kernel library are provided as a convenience for programmers doing SOM development on IBM platforms. These functions are not required to be available in conforming SOM implementations from other vendors. In this category are functions that provide debugging support and formatted stream output. A list of mandatory and optional kernel functions is given in § 5.8, "SOM Kernel Functions", on page 48.

Consult the *SOMObjects Developer Toolkit Programmer's Reference Manual* for more detailed descriptions and specifications of particular kernel functions and methods.

5.1 Addressing, Calling Conventions, and Register Usage

All addresses used in SOM are 32-bit addresses. SOM utilizes the standard system function calling convention defined in § 4, “Procedure Linkage Conventions”, on page 21 for all function and method calls.

5.2 SOM Class Library Structure

A SOM class library contains the implementation of one or more SOM classes. Normally each class in a SOM class library exports at least one function named `<className>NewClass` and two data structures, ClassData named `<className>ClassData` and auxiliary ClassData named `<className>CClassData`. Additional functions and data structures may also be exported but these are not mandated by SOM. We say “normally” because it is possible to construct a SOM class library without these characteristics, but a SOM class without these exports does not offer a client programmer the opportunity to utilize SOM’s *offset* (or static) method resolution mechanism. (The offset method resolution mechanism provides the best performance and is the most commonly employed form of SOM method resolution. Providers of SOM class libraries give client programmers the most options by offering support for offset method resolution and leaving the decision of whether or not to use it to the client programmer.) Typically both the `<className>NewClass` function and the `<className>ClassData` and `<className>CClassData` structures are generated automatically by the SOM Compiler as part of its implementation bindings for a language or produced implicitly by a “Direct-to-SOM” compiler.

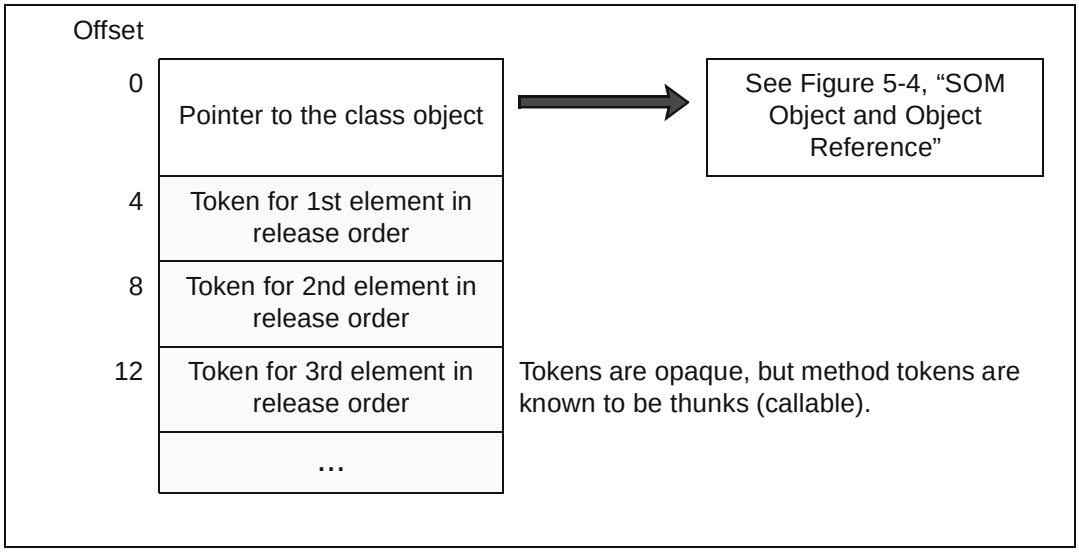
The `<className>NewClass` function constructs, registers with the SOM Class Manager, and returns to the caller, a SOM class object representing the class designated in its name. This function is idempotent. To make it easy to write a `<className>NewClass` function by hand, the SOM Kernel library includes a function named `somBuildClass` which can perform all of these functions based on parameters provided by the caller.

The ClassData structure reflects the **releaseorder** modifier from the implementation section of the class’ IDL specification. The “release order” is a linear ordering of all of the attributes, methods, and public data *introduced* by the class; attributes, methods, or public data items inherited by the class are not included. As newer releases of a class library appear this list may change, but only by extension at the end. Even if a method has been refactored up into one of its ancestor classes (and now appears in that class’ release order), its original position in the release order of the class where it was first introduced is always maintained. Except for the first member of the ClassData structure (which always points to the actual SOM class object) each of its member elements corresponds directly to a release order item and is referred to as a *token*. The tokens are classified as either a method token or a data token. Method tokens are in reality thunks that can be called to perform SOM offset method resolution. They can also be passed to the `somResolve` function (along with a target object) in order to obtain a direct pointer to a method procedure.

The contents of the `<className>ClassData` structure are computed during the execution of the `<className>NewClass` function (commonly by `somBuildClass`) and remain usable for the life of the class. The first element in the ClassData structure is a pointer to the

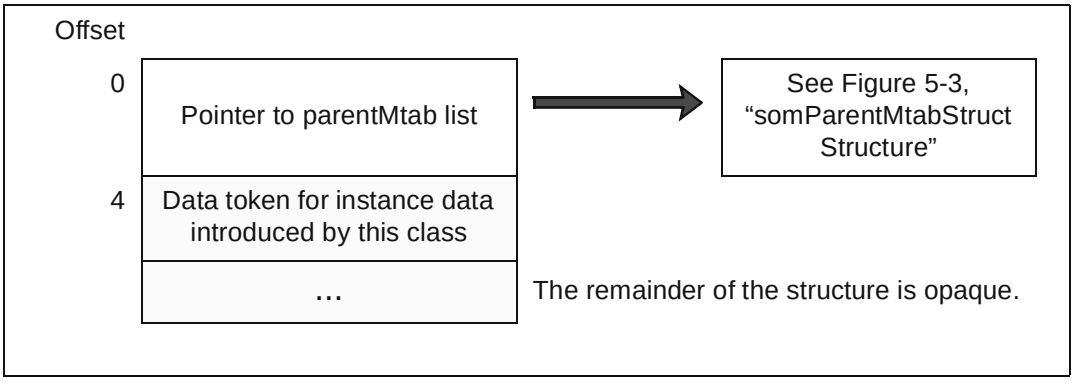
class object (all SOM classes are represented by run-time class objects). The remainder of the ClassData structure has one element for each item in the release order.

Figure 5-1: ClassData Structure



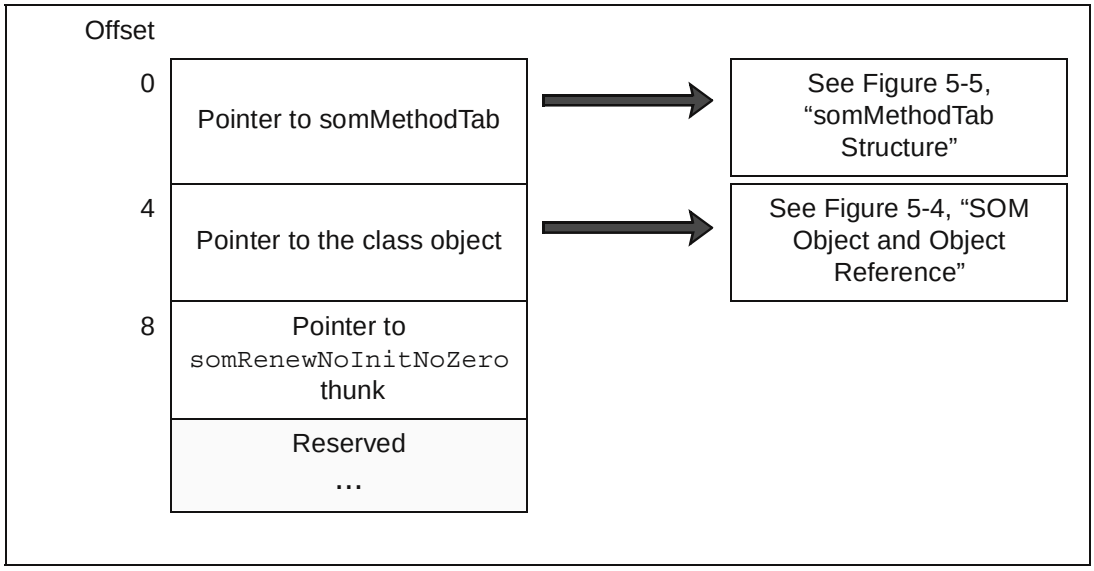
The `<className>CClassData` is an auxiliary structure that is also computed during the construction of a class. (The “C” in the name of this structure is an artifact of its evolution and is not otherwise meaningful). It contains information that may be used by SOM language bindings and “Direct-to-SOM” compilers, such as a list of method table structures for the class and its parent classes. This structure is not generally of direct interest to users of a SOM class since the information in this structure can be obtained dynamically through queries to the class object. By exporting it in this form, more efficient static references can also be built into language bindings.

Figure 5-2: Auxiliary ClassData Structure



The somParentMtabStruct, pointed at by the first element of the auxiliary structure

Figure 5-3: somParentMtabStruct Structure



contains a pointer to a somRenewNoInitNoZero thunk. This thunk provides support for object creation very similar to the somRenewNoInitNoZero method. If the metaclass of the object being created is SOMClass, invoking the thunk results in a fast object instance creation. However, if the metaclass of the object is not SOMClass, the thunk will end up invoking the somRenewNoInitNoZero method. The SOM API requires that somRenewNoInitNoZero always be called when creating a new object whose metaclass is not SOMClass. This is because metaclasses must be guaranteed that they can use somRenewNoInitNoZero to track object creation if this is desired. In either case, a new instance of the receiving class is created by setting the appropriate location in the passed memory block to the receiving class' instance method table. Unlike somNew, no space is allocated. It is the callers responsibility to allocate space which is large enough to hold an instance of the receiving class. The SOM method somGetInstanceSize can be invoked on the class to determine the amount of memory required. A pointer to this allocated space is passed in the call.

A dynamically linked class library is one that can be loaded or unloaded by the SOM class manager (statically linked class libraries cannot be loaded or unloaded, although their classes can be created and destroyed). Each dynamically linked class library is required to provide a special library initialization function (one per library) that can be automatically invoked whenever the library is loaded. One some platforms the operating system will invoke this function. On platforms where the operating system does not provide this capability, the SOM class manager will invoke a function named SOMInitModule which the library is then expected to provide. This library initialization function is responsible for creating and registering class objects for all of the classes in the library.

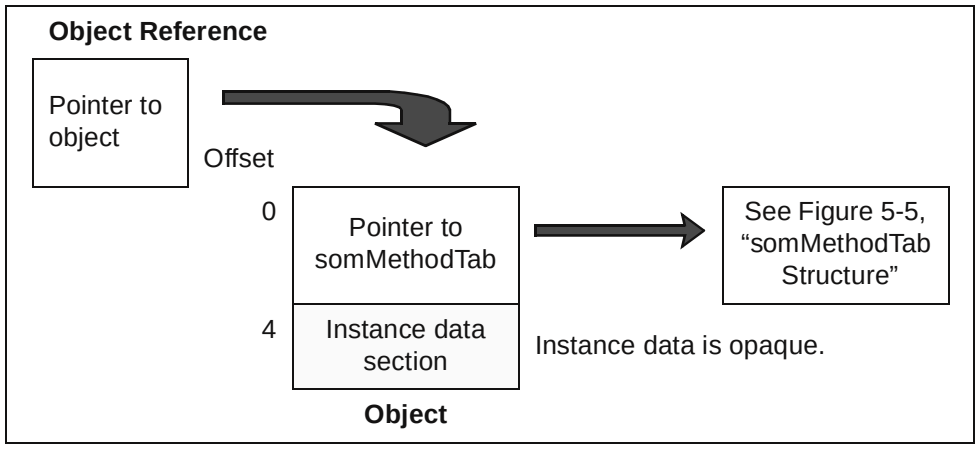
5.3 SOM Objects and Object References

SOM objects are variable in length and semi-opaque. They are composed of two parts: a *method table* pointer and an optional *instance data* section. The method table pointer is found at offset 0 in every SOM object. The instance data section (if it exists) is the opaque part. It is laid out according to the class of the object. Each class that contributes to the derivation of the object's class will have a unique portion of the instance data section, appropriately aligned, to hold its instance data (if any). The layout of this data is known only to the code that implements each of the contributing classes.

SOM object references are simply pointers to SOM objects.

The format of a SOM object and a SOM object reference is illustrated below.

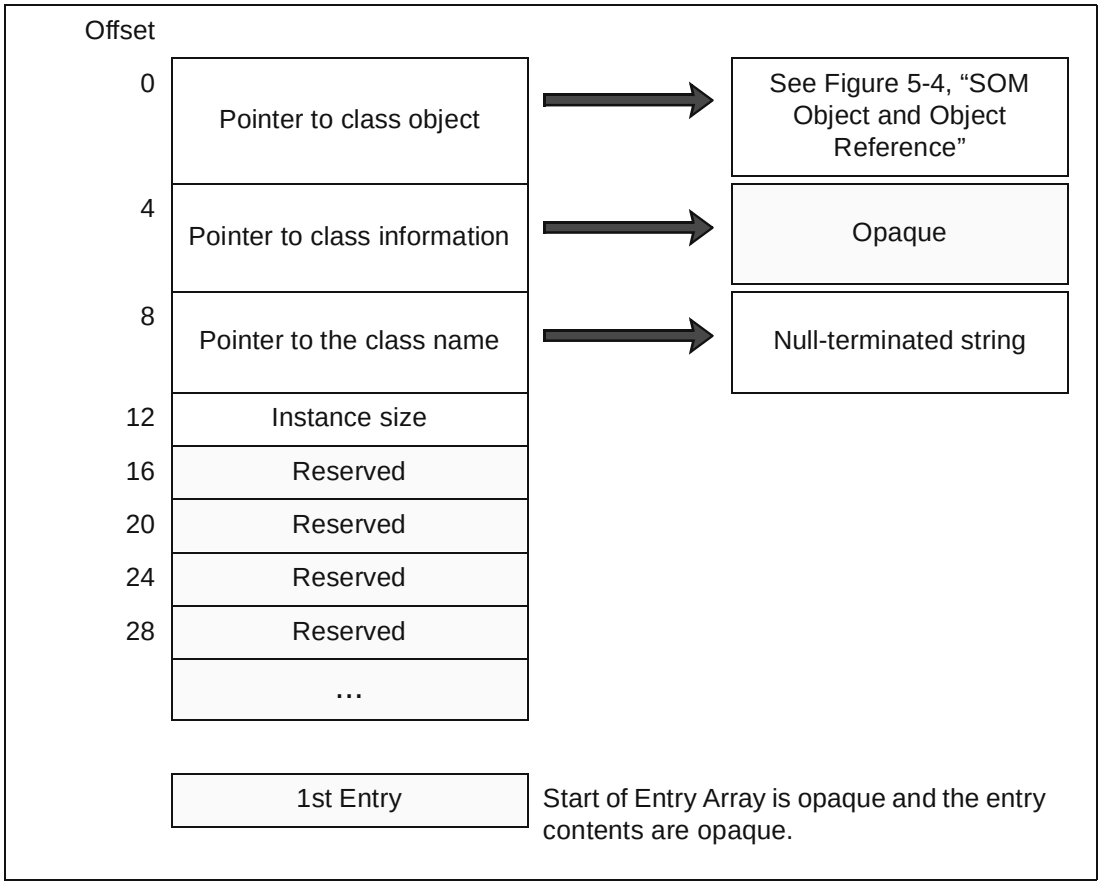
Figure 5-4: SOM Object and Object Reference



5.4 SOM Method Table

All SOM objects point to a data structure referred to as a *method table*. A SOM method table is the principal data structure employed by the offset resolution mechanism. This structure is owned by the class object and created during class initialization. Even classes that do not support offset resolution must provide a method table, although in this case, no entries are needed in the entry section. The SOM method table is semi-opaque. It consists of a fixed section that holds class-related information, and a variable section containing an array of *entries*. Entries are opaque and only used internally by SOM. They may include such things as pointers to method procedures, redispatch stubs, interface identifiers, etc. A block diagram of a SOM method table is shown below.

Figure 5-5: somMethodTab Structure

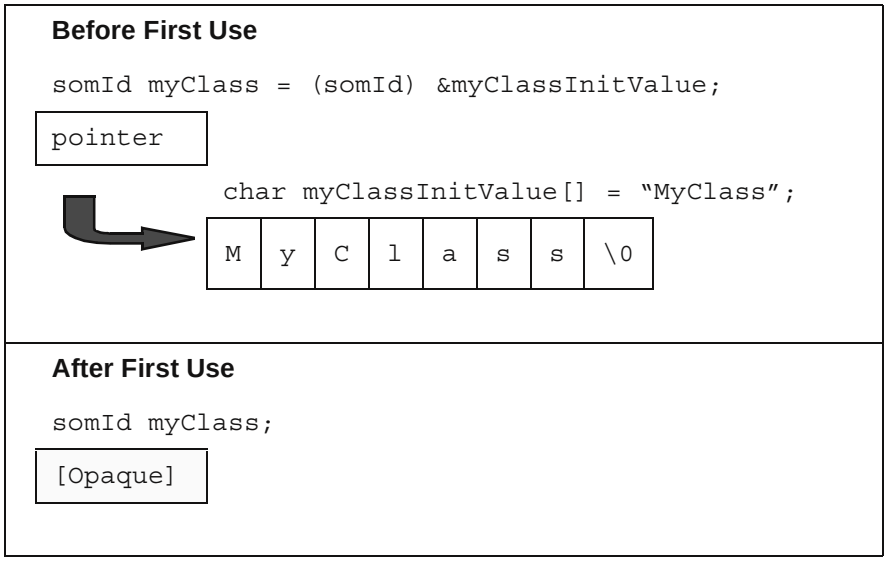


5.5 SOM Ids

Many commonly used methods in the SOM Kernel library refer to class names and method names. To keep these operations as efficient as possible, class names and method names are represented in a special form referred to as a *somId*. The somId form permits very rapid string comparisons.

SomIds can be statically declared as literals in a program or generated dynamically during program execution from strings. A statically declared somId goes through a transformation during its first use which involves registering its value and computing a unique key for it. The

Figure 5-6: Statically Initialized somId



programmer need not be concerned with this transformation as it is performed automatically by any of the SOM Kernel functions that operate on somIds. To subsequently re-extract the string value from a somId, the `somStringFromId` function must be used. SomIds can also be manufactured dynamically using the `somIdFromString` function. SomIds created in this manner must be subsequently freed when they are no longer needed.

If it is known that the strings used to create somIds will exist for the life of the process, then the creation of somIds can be significantly optimized using a *persistent somId bracket*. To start a persistent somId bracket call the SOM Kernel function `somBeginPersistentIds`. Thereafter all subsequent derivations of somIds from strings occurring in the same thread of execution will assume that the string values will persist for the remainder of the process. To end a persistent somId bracket simply call the `somEndPersistentIds` function. Subsequently any somIds manufactured from strings on the same execution thread will not be assumed to persist.

5.6 Basic Operations

In order for any process to make use of SOM, it must first initialize the SOM Kernel by calling the function `somEnvironmentNew` (SOM language bindings typically hide this requirement). This function takes no arguments and is idempotent. It performs any needed SOM initialization and returns an object reference for an instance of the `SOMClassMgr` class. SOM initialization includes the construction of class objects for each of the built-in classes and the instantiation of a single instance of the `SOMClassMgr`. This object can be subsequently used to locate class objects and to load and unload dynamically linked SOM Class libraries. After initialization has completed, the `SOMClassMgrObject` external variable in the SOM Kernel library also contains an object reference for the single instance of the `SOMClassMgr`. This external reference permits compiled programs to refer statically to the SOM class manager.

Once the SOM environment has been initialized, four SOM objects exist. Three of them are class objects; the fourth is the `SOMClassMgrObject`, usually just referred to as the SOM class manager. SOM is now ready to perform basic operations such as subclassing and method resolution. Subclassing involves the creation of a new class and is done by calling the SOM Kernel function `somBuildClass`. The `somBuildClass` function is simply a carefully sequenced invocation of methods provided by the built-in class `SOMClass`, and based on descriptive information passed as arguments to `somBuildClass`.

Since other basic operations such as object instantiation and deinstantiation (freeing objects) are performed by invoking methods on objects (frequently class objects), the remainder of the SOM binary interface reduces to exercising the method resolutions mechanisms.

5.7 Method Resolution Mechanisms

SOM provides three different method resolution mechanisms: offset, name lookup, and dispatch. Implementors of SOM classes decide which of these mechanisms their class will support. If a class supports the offset method resolution mechanism, then a client programmer may use any of the offset, name lookup, or dispatch mechanisms when invoking its methods. If a class supports name lookup but not offset, then a client programmer may use either the name lookup or dispatch mechanism. All SOM classes always support the dispatch resolution mechanism.

Classes that support the offset method resolution mechanism have interfaces declared in IDL and export a `<className>ClassData` structure. Classes that support name lookup, but not offset resolution may have interfaces declared in IDL, but all methods not supporting offset resolution must have the **namelookup** modifier attached to them. Classes that support neither the offset nor the name lookup mechanism do not declare their methods in IDL at all (unless as comments), and need not supply any IDL definition of the class itself, since no static language bindings are provided. Finally, a class may have methods with a mixed level of support. That is, some of its methods would be accessible via the offset, name lookup, or dispatch mechanisms, while others would only be accessible using the name lookup or dispatch mechanisms, with yet other methods only accessible via the dispatch mechanism. All methods of all SOM classes are always accessible via the dispatch mechanism.

Most SOM class implementors provide implementations that support offset method resolution, because it offers the best performance and gives client programmers their choice of any of the three method resolution mechanisms.

5.7.1 Using Offset Method Resolution

The ClassData structure can be used to perform offset method resolution. Offset method resolution is appropriate whenever the class that introduced the method and the name and signature of the method to invoke are all known at compile-time. It offers a highly optimized execution path for invoking a method. In SOM, the method tokens in the ClassData structure are actually method resolution thunks, and invoking a method is achieved by simply calling the appropriate thunk with the arguments needed for the method procedure (the receiving object is always the first argument in this list).

Following is a typical offset method resolution method call.

Figure 5-7: Example Offset Method Resolution Method Call

```
# assume GOT address in r31
   mr    %r3, <ARG1>       # Pointer to receiving object
   mr    %r4, <ARG2>       # argument 2
   mr    %r5, <ARG3>       # argument 3
# call <introducingClassName>ClassData.<methodName>
   lwz   %r11,<introducingClassName>ClassData@got(%r31)
   lwz   %r11,<methodName>(%r11) # offset into ClassData
   mtctr %r11
   bctrl
# r3 now contains the result from the method procedure
```

If it is known that the same method is going to be repeatedly invoked on one or more instances of the same class, a pointer to the actual method procedure selected by offset resolution can be obtained once and the method procedure can then be repeatedly called through the pointer. The SOM Kernel function `somResolve` is used to obtain the method procedure pointer from the target object and method token.

Method procedures may wish to invoke implementations inherited from one or more parent classes. To do so using the offset resolution mechanism, the auxiliary ClassData structure must be used. When a class is derived from other classes, its immediate parent classes are specified using a linear ordering when the `somBuildClass` function is invoked. That ordering is preserved by the `parentMtab` member in the *<className>*`CClassData` structure, which contains a list of method tables for the class and each of its immediate parent classes. For example to invoke an implementation of the method `foo` provided by the 2nd parent class from inside the method procedure for `foo` in class `bar`, the following sequence could be used.

Figure 5-8: Example Offset Method Resolution Parent Method Call

```
# assume GOT address in r31
   lwz   %r3, barCClassData@got(%r31)
   lwz   %r3, parentMtab(%r3) # offset into CClassData
   addi  %r4, 0, 2         # Specify 2nd direct parent class
   lwz   %r5,<introducingClassName>ClassData@got(rGOT)
   lwz   %r5, foo(%r5)     # offset into ClassData
   bl    somParentNumResolve
   mtctr %r3               # parent method procedure pointer
   mr    %r3, <ARG1>       # Pointer to receiving object
   mr    %r4, <ARG2>       # argument 2
   mr    %r5, <ARG3>       # argument 3
   bctrl
# r3 now contains the result from the method procedure
```

5.7.2 Using Name Lookup Method Resolution

Name lookup method resolution is appropriate whenever the method signature is known at compile time but the name of the introducing class or the method name itself is not known at compile time.

The name lookup method resolution mechanism can be exercised in several ways. The easiest way is to call the SOM Kernel function `somResolveByName`, which takes a target object and a method name in the form of a null-terminated string, and returns a method procedure pointer. However, better performance and finer grain control over the name lookup mechanism is available through the use of several class methods provided by the implementation of `SOMClass`. These are `somLookupMethod`, `somFindMethod`, `somFindMethodOK`, `somFindSMethod`, and `somFindSMethodOK`. These methods all represent the method name as a somId, offer different calling sequences, provide different failure modes, and in the case of the `somFindSMethod` limit their resolution to methods which are statically defined. However, since all of these mechanisms are provided as methods themselves, they must be invoked either by the offset resolution mechanism described above, the `somResolveByName` kernel function, or using dispatch resolution,

described in the next section. The quickest way to invoke one of the name lookup methods is, of course, by using offset resolution.

5.7.3 Using Dispatch Method Resolution

The dispatch method resolution mechanism is appropriate whenever the method signature itself is not known at compile time. The name of the method (in the form of a somId), a memory area to hold any result value produced by the method, and a data structure that contains all of the arguments needed for the method are all supplied as arguments to the dispatch mechanism.

The dispatch resolution mechanism is available through the `SOMObject` method `somDispatch`. Because `somDispatch` is implemented as a method it must be invoked using offset or name lookup method resolution (it could also be invoked using the dispatch resolution mechanism, but then you must readdress the question of how you resolve the outer `somDispatch` method).

5.8 SOM Kernel Functions

5.8.1 Required Functions

These functions are required in all conforming SOM implementations.

```
SOMClassMgrNewClass
SOMClassNewClass
SOMObjectNewClass
somAncestorResolve
somApply
somBeginPersistentIds
somBuildClass
somCallInitIfShould
somCheckId
somClassResolve
somCompareIds
somCreateDynamicClass
somDataResolve
somDataResolveChk
somEndPersistentIds
somEnvironmentNew
somExceptionId
somExceptionValue
somExceptionFree
somGetGlobalEnvironment
somIdFromString
somIsObj
somParentNumResolve
somPrintf
somRegisterId
somResolve
somResolveByName
somSetException
somSetExpectedIds
somStringFromId
somTestCls
somTotalRegIds
somUniqueKey
somVprintf
```

5.8.2 Optional Functions

These are optional debugging support functions provided in IBM implementations of SOM.

```
somAssert
somCheckArgs
somLprintf
somPrefixLevel
somTest
```

5.8.3 Obsolete Functions

These are obsolete functions provided for SOM 1.0 compatibility in IBM implementations of SOM.

```
somConstructClass
somParentResolve
```

5.9 SOM Kernel External Variables

5.9.1 Required External Variables

These external variables are required in all conforming SOM implementations.

- Variables that are pointers to user-replaceable functions

```
SOMCalloc
SOMClassInitFuncName
SOMDeleteModule
SOMError
SOMFree
SOMLoadModule
SOMMalloc
SOMOutCharRoutine
SOMRealloc
```

- External data structures

```
SOMClassCClassData
SOMClassClassData
SOMClassMgrCClassData
SOMClassMgrClassData
SOMClassMgrObject
SOMObjectCClassData
SOMObjectClassData
SOM_IdTable
SOM_IdTableSize
SOM_MajorVersion
SOM_MinorVersion
```

5.9.2 Optional External Variables

These are optional debugging support variables found in IBM implementations of SOM.

```
SOM_AssertLevel
SOM_TraceLevel
SOM_WarnLevel
```

5.10 SOM Kernel Class' Release Order

5.10.1 SOMObject

```
somInit
somUninit
somFree
<reserved 1>
somGetClassName
somGetClass
somIsA
somRespondsTo
somIsInstanceOf
somGetSize
somDumpSelf
somDumpSelfInt
somPrintSelf
<reserved 2>
somDispatch
somClassDispatch
```

5.10.2 SOMClassMgr

```
somFindClsInFile
somFindClass
somClassFromId
somRegisterClass
somUnregisterClass
somLocateClassFile
somUnloadClassFile
somGetInitFunction
somMergeInto
somGetRelatedClasses
somSubstituteClass
_get_somInterfaceRepository
_set_somInterfaceRepository
_get_somRegisteredClasses
```

5.10.3 SOMClass

```
somNew
somRenew
somClassReady
somGetName
somDescendedFrom
somCheckVersion
somFindMethod
somFindMethodOK
somSupportsMethod
somGetNumMethods
somGetInstanceSize
somGetInstancePartSize
somGetMethodIndex
somGetNumStaticMethods
somGetPclsMtab
somGetClassMtab
somAddStaticMethod
somOverrideSMethod
somAddDynamicMethod
somFindSMethod
```

```
somFindSMethodOK
somGetMethodDescriptor
somGetNthMethodInfo
somSetClassData
somGetClassData
somNewNoInit
somRenewNoInit
somGetInstanceToken
somGetMemberToken
somSetMethodDescriptor
somGetMethodData
somOverrideMtab
somGetMethodToken
somGetParents
somGetPclsMtabs
somInitMIClass
somGetVersionNumbers
somLookupMethod
_get_somInstanceDataOffsets
somRenewNoZero
somRenewNoInitNoZero
somAllocate
somDeallocate
somGetRdStub
somGetNthMethodData
```

6 System Object Exception Handling

This chapter will explain the model and interfaces to support system object exception handling. This chapter will be provided in a future release of this document.

7 Execution Model

This chapter discusses the execution model for programs conforming to this ABI. It also provides code examples demonstrating function tags and how fundamental operations such as function calls and data access can be performed.

7.1 Code Model

The code model supported by this ABI is the Position Independent Code (PIC) model. Instructions in this code model generally hold only relative addresses and not absolute addresses. PIC code segments can be loaded at any virtual address in memory and execute properly. Code segments containing absolute addresses will only execute properly if loaded at a specific virtual address making the absolute addresses contained in the instructions coincide with the instruction's virtual addresses. This ABI specifies that both executables and dynamic link libraries shall contain position independent code. The use of absolute code (*i.e.* position dependent code) is not supported by the ABI and is non-conforming.

System V ABI Note: The *System V Application Binary Interface, PowerPC Processor Supplement* specifies that executables use the absolute code model. This ABI specifies that executables use the position independent code model.

When the system creates the process image from an executable and various dynamic link libraries, the system chooses the virtual addresses at which the load segments in the load modules will be loaded. With position independent code, the system loader can load the code segments at any virtual address and the code will function correctly.

Position independent code relies on two techniques.

- Instructions which transfer control (*e.g.* branch instructions) either contain an address relative to the virtual address of the instruction or use registers to hold the target address. A relative branch instruction computes its target address in terms of a positive or negative displacement from the virtual address of the instruction.
- When an absolute address is required, the program must compute it. Rather than embedding absolute addresses in the instruction stream, the compiler emits code that computes the needed address during execution.

The architecture of the PowerPC processor provides for both relative branch instructions and branch instructions that utilize a target address contained in a register. This provides support for the first technique above.

The Global Offset Table (GOT) is used in support of the second technique. The Global Offset Table holds the absolute addresses that cannot be held in position independent code. The position independent code obtains the addresses of memory objects from the GOT and can then access the objects. Each load module that requires such an absolute address will contain a GOT. The GOT for each load module is relocated by the system when the process image is built to contain the proper absolute virtual addresses. See § 11.5, “Global Offset Table (GOT)”, on page 135 for more details.

Additionally, the Procedure Linkage Table (PLT) is used to redirect calls to functions outside the load module. Since the absolute address of the target function is not known until the process image is created, the PLT is used as an intermediary to transfer control between load modules. Each load module that calls functions external to the load module will contain a PLT. The PLT for each load module is updated by the system when the process image is built to contain the proper instructions to transfer control to the absolute address of the target function. See § 11.6, “Procedure Linkage Table (PLT)”, on page 137 for more details.

The use of the PLT does not affect the code generated to make functions calls. The same code is used for intra-module and inter-module calls with the static linker redirecting calls to the PLT as necessary. However, the use of the GOT does have impact on the code generated to access data objects. Functions must contain code to establish addressability to the GOT and code to access the GOT to obtain the desired absolute addresses. Example code can be found in § 7.3, “Code Examples”.

Due to the processor’s architecture, there are two position independent code models: “small model” and “large model”. The choice between the models is based upon a trade-off between the efficiency of the code and the supported size of the Global Offset Table. The small model supports a GOT up to 64K bytes in size (16384 entries) and has more efficient code than the large model. If support for a GOT with greater than 16384 entries is required, then the more general and less efficient large model should be used. Where applicable, code examples of both models will be shown in § 7.3, “Code Examples”.

7.2 Function Tags

Function tags provide a means of determining the contents of non-volatile registers as they were when a function was entered. Each function shall be immediately preceded by a function tag word. Given the address of the next instruction to be executed, an exception handler or debugger can search the text towards lower addresses looking for a function tag. Using the information in the function tag and the stack frame layout (specified in § 4.2, “Stack Frames”, on page 25), the contents of the non-volatile registers at function entry can be determined.

Function tag words are recognized by having bits `0-5` set to zero, which makes the word a reserved or illegal instruction. A special degenerative form of the function tag word is recognized. This is a zero word. If a compiler (or assembly programmer) does not support function tag words as described in this section, then this special form of the tag word shall be placed immediately before the function. This enables an exception handler or debugger to locate this tag word and recognize that it is not possible to determine the contents of the non-volatile registers as they were at function entry. Appropriate action can then be taken.

Note: All functions shall be preceded by a function tag word. The function tag word shall be immediately followed by the first instruction of the function. The function tag word shall be either the function tag word described in this section or the special degenerative form of the function tag word (a zero word).

The following table describes the format of the function tag word. Bit 0 is the most significant bit in the word and all tag words are stored using the data encoding specified in the ELF header.

Table 7-1: Function Tag Word

Field Name	Bit(s)	Field Description
`identifier`	`0-5`	This field identifies the word as a function tag and contains the value zero.
`version`	`6-7`	This field contains the function tag version number. The current version is zero.
`range`	`8-15`	This field contains the number of words (instructions) between the function tag and the first address at which the stack frame has been acquired and the non-volatile registers specified by this tag have been saved. At the end of this interval, the non-volatile registers still contain the values they had at function entry. **Note:** This field may not contain zero unless either `long_form` is one or `lr_inreg` is one.

Table 7-1: Function Tag Word (Continued)

Field Name	Bit(s)	Field Description
long_form	16	This bit specifies whether the function tag has long form function tag information. If this bit is zero, then the function tag does not have long form function tag information. If this bit is one, then the next lower-addressed word than this function tag word contains a signed offset from the address of the function tag word to the long form function tag information for this function (in the .tag section). See § 7.2.1, “Long Form Function Tag Information”, on page 60 for details of the long form tag information.
token_present	17	This bit specifies if a compiler-specific token is present. This token may contain any value and can be used by the compiler for any reason. If this bit is zero, then no token is present. If this bit is one, then a token is present. If long_form is zero, then the next lower-addressed word than this function tag word contains the token. If long_form is one, then the token is contained in the next lower-addressed word than the long form signed offset for this function tag word.
gpr31_nosave	18	This bit specifies if r31 participates in the saving of non-volatile GPRs. If this bit is zero, then space for r31 is allocated in the GPR save area of the stack frame (if non-volatile GPRs are saved). If this bit is one, then the function does not modify r31 and no space for r31 is allocated in the GPR save area of the stack frame (if non-volatile GPRs are saved) and r31 is not saved. When this bit is set, r30 is stored at the highest addressed word of the GPR save area of the stack frame.
alloca_gpr30	19	This bit specifies if the function uses r30 to hold the frame pointer to the local variable area of the stack frame. This may occur if dynamic stack space allocation is used. (See § 7.3.4, “Dynamic Stack Space Allocation”, on page 74.) If this bit is zero, then r30 does not hold the frame pointer. If this bit is one, then r30 holds the frame pointer to the local variable area of the stack frame.
lr_inreg	20	This bit specifies where the value of LR at function entry can be found. If this bit is zero, then the LR save word of the previous stack frame contains the value of LR as it was at function entry. If this bit is one, then LR contains the value of LR as it was at function entry.

Table 7-1: Function Tag Word (Continued)

Field Name	Bit(s)	Field Description
cr_saved	21	This bit specifies if CR has been saved in the CR save word of the stack frame. If this bit is zero, then CR has not been saved in the CR save word of the stack frame If this bit is one, then the CR save word of the stack frame contains the value of CR as it was at function entry.
fpr_space	22-26	This field contains the size in doublewords of the FPR save area of the stack frame. If long_form is zero, then all the non-volatile FPRs implied by this field must be saved by the end of the range of this function tag. If long_form is one, then the long form function tag information specifies the ranges where the non-volatile FPRs are saved. This field merely indicates the size of the FPR save area in the stack frame.
gpr_space	27-31	This field contains the size in words of the GPR save area of the stack frame. If long_form is zero, then all the non-volatile GPRs implied by this field (and gpr31_nosave) must be saved by the end of the range of this function tag. If long_form is one, then the long form function tag information specifies the ranges where the non-volatile GPRs are saved. This field merely indicates the size of the GPR save area in the stack frame.

7.2.1 Long Form Function Tag Information

Functions which intersperse saving some non-volatile registers with using other non-volatile registers (*i.e.* functions that do not save all used non-volatile registers in the function prologue and delay saving some or all used non-volatile registers until later in the function) will need to use the long form function tag information to specify when the non-volatile registers have been saved in the stack frame. The long form function tag information is not stored in-line with the program instructions like the function tag words but is stored in the .tags section and the in-line function tag has a signed offset to the long form function information. The long form function tag information for a function consists of a sequence of entries, each two words in size, describing a range of instructions in the function.

The following tables describe the two-word long form function tag information entries. The first word, word 1, is the lowest addressed word of the two-word entry. Bit 0 is the most significant bit in the word and all words are stored using the data encoding specified in the ELF header.

Table 7-2: Long Form Function Tag Information, Word 1

Field Name	Bit(s)	Field Description
range	0-13	This field contains the number of words (instructions) between the end of the previous range (in the function tag word, if this is the first long form function tag information entry, or in the previous long form function tag information entry) and the first address at which the non-volatile registers specified by this tag have been saved. At the end of this range, the non-volatile registers that were saved during this range still contain the values they had at function entry.
fpr_saved	14-31	This field contains one bit for each non-volatile FPR. The most significant bit represents f14 and the least significant bit represents f31. If a bit is zero, then the corresponding FPR has not been saved in the appropriate place in the FPR save area of the stack frame. If a bit is one, then the corresponding FPR has been saved in the appropriate place in the FPR save area of the stack frame.

Table 7-3: Long Form Function Tag Information, Word 2

Field Name	Bit(s)	Field Description
last_entry	0	This bit specifies if this entry is the last entry in the sequence of long form function tag information entries for a function. If this bit is zero, then this entry is immediately followed by another entry for this function. If this bit is one, then this entry is the last entry for this function.
reserved	1-6	These bits are reserved and shall contain zero.
alloca_gpr	7-11	This field specifies if the function uses a GPR to hold a frame pointer to the local variable area of the stack frame. This may occur if dynamic stack space allocation is used. (See § 7.3.4, “Dynamic Stack Space Allocation”, on page 74.) If this field is zero, then there is no frame pointer. If this field is non-zero, then the field holds the register number of the GPR used as the frame pointer to the local variable area of the stack frame.

Table 7-3: Long Form Function Tag Information, Word 2 (Continued)

Field Name	Bit(s)	Field Description
lr_inreg	12	This bit specifies where the value of LR at function entry can be found. If this bit is zero, then the LR save word of the previous stack frame contains the value of LR as it was at function entry. If this bit is one, then LR contains the value of LR as it was at function entry.
cr_saved	13	This bit specifies if CR has been saved in the CR save word of the stack frame. If this bit is zero, then CR has not been saved in the CR save word of the stack frame If this bit is one, then the CR save word of the stack frame contains the value of CR as it was at function entry.
gpr_saved	14-31	This field contains one bit for each non-volatile GPR. The most significant bit represents r14 and the least significant bit represents r31. If a bit is zero, then the corresponding GPR has not been saved in the appropriate place in the GPR save area of the stack frame. If a bit is one, then the corresponding GPR has been saved in the appropriate place in the GPR save area of the stack frame.

7.3 Code Examples

The code examples in this section illustrate how operations may be done but not how they must be done. The examples will be a mixture of ANSI C and PowerPC assembler code but programming languages other than ANSI C may choose to use the same conventions demonstrated in the examples. However, failure to do so does not prevent a program from conforming to the ABI.

The examples below are simplified code fragments and are not meant to show optimal code sequences or reproduce compiler-generated code. They are intended to explain the execution model and demonstrate concepts.

7.3.1 Function Prologue and Epilogue

This section provides code examples for function prologues and epilogues. A function prologue acquires a stack frame if necessary, and saves any non-volatile registers used by the function. A function epilogue “unwinds” the work done by the prologue by restoring the non-volatile registers and releasing the stack frame before returning to the caller. See § 4.2, “Stack Frames”, on page 25 for the rules regarding stack frames.

A compiler may generate in-line code to save and restore the non-volatile registers used by the function. However, if many non-volatile registers must be preserved, it may be more efficient for the compiler to generate a call to a compiler-supplied support routine which performs the saves or restores. Following are some examples of support routines which perform these functions.

Note: The routines below rely on an address passed in the volatile register `r11`. Therefore these routines must be statically-linked into the same load module as the caller.

The `_savefpr_n_l` routines save the non-volatile FPRs *n* through 31 in the new stack frame and `LR` in the previous stack frame. The routines take two parameters.

- `r0` shall contain the value of `LR` at entry to the function whose prologue called this routine.
- `r11` shall contain the address of the word immediately following the FPR save area. This is also the address of the previous stack frame.

Figure 7-1: `_savefpr_n_l` Routines

```
                .word 0 # tag word
_savefpr_14_l:  stfd  %f14, -144(%r11)
_savefpr_15_l:  stfd  %f15, -136(%r11)
_savefpr_16_l:  stfd  %f16, -128(%r11)
_savefpr_17_l:  stfd  %f17, -120(%r11)
_savefpr_18_l:  stfd  %f18, -112(%r11)
_savefpr_19_l:  stfd  %f19, -104(%r11)
_savefpr_20_l:  stfd  %f20, -96(%r11)
_savefpr_21_l:  stfd  %f21, -88(%r11)
_savefpr_22_l:  stfd  %f22, -80(%r11)
_savefpr_23_l:  stfd  %f23, -72(%r11)
_savefpr_24_l:  stfd  %f24, -64(%r11)
_savefpr_25_l:  stfd  %f25, -56(%r11)
_savefpr_26_l:  stfd  %f26, -48(%r11)
_savefpr_27_l:  stfd  %f27, -40(%r11)
_savefpr_28_l:  stfd  %f28, -32(%r11)
_savefpr_29_l:  stfd  %f29, -24(%r11)
_savefpr_30_l:  stfd  %f30, -16(%r11)
_savefpr_31_l:  stfd  %f31, -8(%r11)
                stw   %r0, 4(%r11)
                blr
```

The `_restfpr_n_l` routines restore the non-volatile FPRs *n* through 31, restore `LR`, release the stack frame and return to the caller of the function whose epilogue branched to this routine. The routines take one parameter.

- `r11` shall contain the address of the word immediately following the FPR save area. This is also the address of the previous stack frame.

Figure 7-2: `_restfpr_n_l` Routines

```
                .word 0 # tag word
_restfpr_14_l:  lfd   %f14, -144(%r11)
_restfpr_15_l:  lfd   %f15, -136(%r11)
_restfpr_16_l:  lfd   %f16, -128(%r11)
_restfpr_17_l:  lfd   %f17, -120(%r11)
_restfpr_18_l:  lfd   %f18, -112(%r11)
_restfpr_19_l:  lfd   %f19, -104(%r11)
_restfpr_20_l:  lfd   %f20, -96(%r11)
_restfpr_21_l:  lfd   %f21, -88(%r11)
_restfpr_22_l:  lfd   %f22, -80(%r11)
_restfpr_23_l:  lfd   %f23, -72(%r11)
_restfpr_24_l:  lfd   %f24, -64(%r11)
_restfpr_25_l:  lfd   %f25, -56(%r11)
_restfpr_26_l:  lfd   %f26, -48(%r11)
_restfpr_27_l:  lfd   %f27, -40(%r11)
_restfpr_28_l:  lfd   %f28, -32(%r11)
_restfpr_29_l:  lfd   %f29, -24(%r11)
_restfpr_30_l:  lfd   %f30, -16(%r11)
_restfpr_31_l:  lwz   %r0, 4(%r11)
                lfd   %f31, -8(%r11)
                mtlr  %r0
                ori   %r1, %r11, 0
                blr
```

There are two versions of the non-volatile GPR save and restore routines. The `_savegpr_n_l` and `_restgpr_n_l` routines are used when no non-volatile FPRs need to be preserved by the function. The `_savegpr_n` and `_restgpr_n` routines are used when non-volatile FPRs need to be preserved by the function and `_savefpr_n_l` and `_restfpr_n_l` routines are being used.

The `_savegpr_n_l` routines save the non-volatile GPRs *n* through 31 in the new stack frame and `LR` in the previous stack frame. The routines take two parameters.

- `r0` shall contain the value of `LR` at entry to the function whose prologue called this routine.
- `r11` shall contain the address of the word immediately following the GPR save area. Since there is no FPR save area in the stack frame, the address in `r11` is also the address of the previous stack frame.

Figure 7-3: `_savegpr_n_l` Routines

```
                .word 0 # tag word
_savegpr_14_l:  stw   %r14, -72(%r11)
_savegpr_15_l:  stw   %r15, -68(%r11)
_savegpr_16_l:  stw   %r16, -64(%r11)
_savegpr_17_l:  stw   %r17, -60(%r11)
_savegpr_18_l:  stw   %r18, -56(%r11)
_savegpr_19_l:  stw   %r19, -52(%r11)
_savegpr_20_l:  stw   %r20, -48(%r11)
_savegpr_21_l:  stw   %r21, -44(%r11)
_savegpr_22_l:  stw   %r22, -40(%r11)
_savegpr_23_l:  stw   %r23, -36(%r11)
_savegpr_24_l:  stw   %r24, -32(%r11)
_savegpr_25_l:  stw   %r25, -28(%r11)
_savegpr_26_l:  stw   %r26, -24(%r11)
_savegpr_27_l:  stw   %r27, -20(%r11)
_savegpr_28_l:  stw   %r28, -16(%r11)
_savegpr_29_l:  stw   %r29, -12(%r11)
_savegpr_30_l:  stw   %r30, -8(%r11)
_savegpr_31_l:  stw   %r31, -4(%r11)
                stw   %r0, 4(%r11)
                blr
```

The `_restgpr_n_l` routines restore the non-volatile GPRs *n* through 31, restore `LR`, release the stack frame and return to the caller of the function whose epilogue branched to this routine. The routines take one parameter.

- `r11` shall contain the address of the word immediately following the GPR save area. Since there is no FPR save area in the stack frame, the address in `r11` is also the address of the previous stack frame.

Figure 7-4: `_restgpr_n_l` Routines

```
                .word 0 # tag word
_restgpr_14_l:  lwz   %r14, -72(%r11)
_restgpr_15_l:  lwz   %r15, -68(%r11)
_restgpr_16_l:  lwz   %r16, -64(%r11)
_restgpr_17_l:  lwz   %r17, -60(%r11)
_restgpr_18_l:  lwz   %r18, -56(%r11)
_restgpr_19_l:  lwz   %r19, -52(%r11)
_restgpr_20_l:  lwz   %r20, -48(%r11)
_restgpr_21_l:  lwz   %r21, -44(%r11)
_restgpr_22_l:  lwz   %r22, -40(%r11)
_restgpr_23_l:  lwz   %r23, -36(%r11)
_restgpr_24_l:  lwz   %r24, -32(%r11)
_restgpr_25_l:  lwz   %r25, -28(%r11)
_restgpr_26_l:  lwz   %r26, -24(%r11)
_restgpr_27_l:  lwz   %r27, -20(%r11)
_restgpr_28_l:  lwz   %r28, -16(%r11)
_restgpr_29_l:  lwz   %r29, -12(%r11)
_restgpr_30_l:  lwz   %r30, -8(%r11)
_restgpr_31_l:  lwz   %r0, 4(%r11)
                lwz   %r31, -4(%r11)
                mtlr  %r0
                ori   %r1, %r11, 0
                blr
```

After using a `_savefpr_n_l` routine, a `_savegpr_n` routine can be used to save non-volatile GPRs. The `_savegpr_n` routines save the non-volatile GPRs *n* through 31 in the new stack frame. The routines take one parameter.

- `r11` shall contain the address of the word immediately following the GPR save area.

Figure 7-5: `_savegpr_n` Routines

```
                .word 0 # tag word
_savegpr_14:    stw   %r14, -72(%r11)
_savegpr_15:    stw   %r15, -68(%r11)
_savegpr_16:    stw   %r16, -64(%r11)
_savegpr_17:    stw   %r17, -60(%r11)
_savegpr_18:    stw   %r18, -56(%r11)
_savegpr_19:    stw   %r19, -52(%r11)
_savegpr_20:    stw   %r20, -48(%r11)
_savegpr_21:    stw   %r21, -44(%r11)
_savegpr_22:    stw   %r22, -40(%r11)
_savegpr_23:    stw   %r23, -36(%r11)
_savegpr_24:    stw   %r24, -32(%r11)
_savegpr_25:    stw   %r25, -28(%r11)
_savegpr_26:    stw   %r26, -24(%r11)
_savegpr_27:    stw   %r27, -20(%r11)
_savegpr_28:    stw   %r28, -16(%r11)
_savegpr_29:    stw   %r29, -12(%r11)
_savegpr_30:    stw   %r30, -8(%r11)
_savegpr_31:    stw   %r31, -4(%r11)
                blr
```

Before using a _restfpr_*n*_l routine, a _restgpr_*n* routine can be used to restore non-volatile GPRs. The _restgpr_*n* routines restore the non-volatile GPRs *n* through 31 from the stack frame. The routines take one parameter.

- r11 shall contain the address of the word immediately following the GPR save area.

Figure 7-6: _restgpr_*n* Routines

```
                .word 0 # tag word
_restgpr_14:    lwz   %r14, -72(%r11)
_restgpr_15:    lwz   %r15, -68(%r11)
_restgpr_16:    lwz   %r16, -64(%r11)
_restgpr_17:    lwz   %r17, -60(%r11)
_restgpr_18:    lwz   %r18, -56(%r11)
_restgpr_19:    lwz   %r19, -52(%r11)
_restgpr_20:    lwz   %r20, -48(%r11)
_restgpr_21:    lwz   %r21, -44(%r11)
_restgpr_22:    lwz   %r22, -40(%r11)
_restgpr_23:    lwz   %r23, -36(%r11)
_restgpr_24:    lwz   %r24, -32(%r11)
_restgpr_25:    lwz   %r25, -28(%r11)
_restgpr_26:    lwz   %r26, -24(%r11)
_restgpr_27:    lwz   %r27, -20(%r11)
_restgpr_28:    lwz   %r28, -16(%r11)
_restgpr_29:    lwz   %r29, -12(%r11)
_restgpr_30:    lwz   %r30, -8(%r11)
_restgpr_31:    lwz   %r31, -4(%r11)
                blr
```

Note: The Load/Store Multiple instructions should not be used because they are generally slower than a collection of individual register loads/stores and they also cause alignment exceptions in the Little Endian mode of the processor.

If any of the non-volatile fields of CR are used, then CR must also be saved by the prologue and restored by the epilogue.

A function that makes references to static data will need to establish addressability to the Global Offset Table. If static data references are only made within conditional code, then establishing addressability to the Global Offset Table can be deferred until it is known to be needed. See § 11.5, “Global Offset Table (GOT)”, on page 135 for more information.

The following sample code demonstrates full non-volatile FPR and GPR saves, a CR save and establishing addressability to the Global Offset Table. A stack frame size of less than 32K bytes is assumed.

Figure 7-7: Function Prologue and Epilogue Sample Code (Small Stack)

```
           .word 0x00080652       # tag word
function:  ori   %r11, %r1, 0     # r11=address previous frame
           stwu  %r1, -240(%r1)   # acquire new frame
           mflr  %r0              # set r0 to lr
           bl    _savefpr_14_l    # save lr, f14-f31
           addi  %r11, %r11, -144 # set r11 for _savegpr
           bl    _savegpr_14      # save r14-r31
           mfcr  %r0              # set r0 to cr
           stw   %r0, -76(%r11)   # save cr
           ...
           bl    _GLOBAL_OFFSET_TABLE_-4
           mflr  %r31             # r31=GOT reference address
           ...
           addi  %r11, %r1, 96    # set r11 for _restgpr
           lwz   %r0, -76(%r11)   # set r0 to original cr
           mtcrf 0x38, %r0        # restore non-volatile fields
           bl    _restgpr_14      # restore r14-r31
           addi  %r11, %r1, 240   # r11=address previous frame
           b     _restfpr_14_l    # restore f14-f31, lr and ret
```

The following sample code demonstrates acquiring a stack frame greater than 32K bytes in size, an in-line save of non-volatile GPRs and establishing addressability to the Global Offset Table.

Figure 7-8: Function Prologue and Epilogue Sample Code (Large Stack)

```
           .word 0x00080002       # tag word
function:  ori   %r11, %r1, 0     # r11=address previous frame
           addis %r12, 0, -1      # set r12 to the complement of
           addi  %r12, %r12, 65536-40016 # stack frame size(40016)
           stwux %r1, %r1, %r12   # acquire new frame
           mflr  %r0              # set r0 to lr
           stw   %r30, -8(%r11)   # save r30
           stw   %r31, -4(%r11)   # save r31
           stw   %r0, 4(%r11)     # save lr in previous frame
           ori   %r30, %r11, 0    # r30=address previous frame
           ...
           bl    _GLOBAL_OFFSET_TABLE_-4
           mflr  %r31             # r31=GOT reference address
           ...
           ori   %r11, %r30, 0    # r11=address previous frame
           lwz   %r0, 4(%r11)     # set r0 to lr
           lwz   %r30, -8(%r11)   # restore r30
           lwz   %r31, -4(%r11)   # restore r31
           mtlr  %r0              # restore lr
           ori   %r1, %r11, 0     # release stack frame
           blr                    # return to caller
```

7.3.2 Static Data Access

This section demonstrates accessing data objects with static storage duration. Instructions in the PowerPC Architecture cannot hold 32-bit addresses and position independent code does not contain absolute addresses. So in order to access a static data object, the Global Offset Table entry for the static data object is referenced to get its absolute address. An automatic data object, which is resident on the stack, can be accessed relative to the stack frame pointer.

In order to locate an entry in the GOT, two pieces of information are needed. First the reference address of the GOT needs to be computed and second the offset of the GOT entry from the reference address must be known. See § 7.3.1, “Function Prologue and Epilogue”, above for example code that establishes the reference address of the GOT. The offset of the GOT entry is supplied by the static linker when the load module is built.

In the examples in this section, we will assume that the reference address of the GOT has already been loaded into `r31`. The assembly syntax *`symbol`*`@got` denotes the offset of the GOT entry for the symbol *`symbol`* from the GOT reference address. This assumes the “small” code model where the offset can be represented as a signed 16-bit value. For “large” code model, where offsets may be larger, the assembly syntax *`symbol`*`@got@ha`, *`symbol`*`@got@h`, and *`symbol`*`@got@l` is used to denote the high-adjusted 16-bits, the high 16-bits and the low 16-bits of the offset of the GOT entry for the symbol *`symbol`* from the

GOT reference address. (See § 9.1.2.3.1, “Relocation Types”, on page 106 for a mathematical description of `@ha`, `@h` and `@l`.)

Figure 7-9: Static Data Access (Small Model)

```
extern int src;
extern int dst;
extern int *ptr;

dst = src;

ptr = &dst;

*ptr = src;
```

```
# Assume r31 contains the GOT reference
# address

   lwz   %r11, src@got(%r31)
   lwz   %r12, dst@got(%r31)
   lwz   %r0, 0(%r11)
   stw   %r0, 0(%r12)

   lwz   %r0, dst@got(%r31)
   lwz   %r12, ptr@got(%r31)
   stw   %r0, 0(%r12)

   lwz   %r11, src@got(%r31)
   lwz   %r12, ptr@got(%r31)
   lwz   %r0, 0(%r11)
   lwz   %r12, 0(%r12)
   stw   %r0, 0(%r12)
```

Figure 7-10: Static Data Access (Large Model)

```
extern int src;
extern int dst;
extern int *ptr;

dst = src;

ptr = &dst;

*ptr = src;
```

```
# Assume r31 contains the GOT reference
# address

   addis %r11, %r31, src@got@ha
   lwz   %r11, src@got@l(%r11)
   addis %r12, %r31, dst@got@ha
   lwz   %r12, dst@got@l(%r12)
   lwz   %r0, 0(%r11)
   stw   %r0, 0(%r12)

   addis %r11, %r31, dst@got@ha
   lwz   %r0, dst@got@l(%r11)
   addis %r12, %r31, ptr@got@ha
   lwz   %r12, ptr@got@l(%r12)
   stw   %r0, 0(%r12)

   addis %r11, %r31, src@got@ha
   lwz   %r11, src@got@l(%r11)
   addis %r12, %r31, ptr@got@ha
   lwz   %r12, ptr@got@l(%r12)
   lwz   %r0, 0(%r11)
   lwz   %r12, 0(%r12)
   stw   %r0, 0(%r12)
```

7.3.3 Function Calls

This section demonstrates making direct and indirect (through a pointer) function calls. The `bl` instruction is used to make direct function calls. The `bl` instruction is a self-relative branch instruction with a target branch displacement of ±32MB. This provides a potential limit on the size of load modules (and also on the location of the Global Offset Table and Procedure Linkage Table).

The `bl` instruction can be used to call functions in the same load module or in a different load module. If the function is in a different load module, then the static linker will redirect the `bl` instruction to a PLT entry which the dynamic linker will arrange to point to the target function.

Figure 7-11: Direct Function Call

```
extern void func();

func();
```

```
   bl    func
```

For indirect function calls, a `bctrl` (or `blrl`) instruction can be used.

Figure 7-12: Indirect Function Call (Small Model)

```
extern void func();
extern void (*ptr)();

ptr = func;



(*ptr)();
```

```
# Assume r31 contains the GOT reference
# address

   lwz   %r0, func@got(%r31)
   lwz   %r12, ptr@got(%r31)
   stw   %r0, 0(%r12)

   lwz   %r12, ptr@got(%r31)
   lwz   %r0, 0(%r12)
   mtctr %r0
   bctrl
```

Figure 7-13: Indirect Function Call (Large Model)

```
extern void func();
extern void (*ptr)();

ptr = func;




(*ptr)();
```

```
# Assume r31 contains the GOT reference
# address

   addis %r11, %r31, func@got@ha
   lwz   %r0, func@got@l(%r11)
   addis %r12, %r31, ptr@got@ha
   lwz   %r12, ptr@got@l(%r12)
   stw   %r0, 0(%r12)

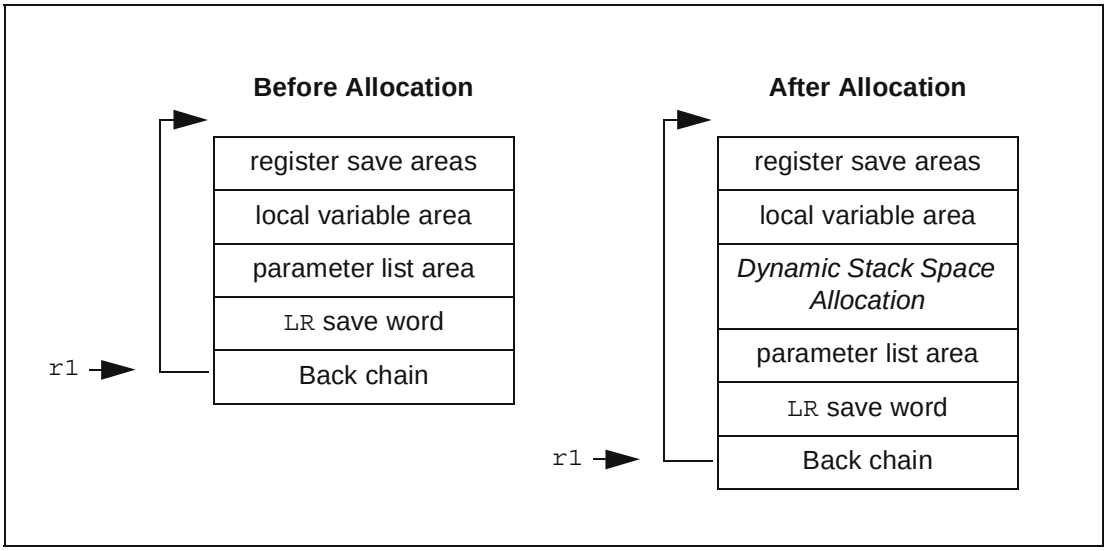
   addis %r12, %r31, ptr@got@ha
   lwz   %r12, ptr@got@l(%r12)
   lwz   %r0, 0(%r12)
   mtctr %r0
   bctrl
```

7.3.4 Dynamic Stack Space Allocation

The C language does not require dynamic stack space allocation within a stack frame. Stack frames are dynamically allocated on the stack with individual frames having static sizes, based upon program execution. For other languages that need dynamic stack space allocation, it is supported by the architecture and the procedure linkage conventions. Thus functions written in these languages can call C functions and vice versa.

Figure 7-14, “Dynamic Stack Space Allocation”, shows the stack frame before and after dynamic stack space allocation. Dynamic stack space allocation is accomplished by “opening” the stack frame between the local variable area and the parameter list area.

Figure 7-14: Dynamic Stack Space Allocation



The register save area and the local variable area do not change in size or position as a result of the dynamic stack space allocation. The parameter list area is required by the procedure linkage conventions to be at a fixed offset from the stack pointer (`r1`), so this area must move when dynamic stack space allocation occurs. When this occurs, the offsets from the stack pointer to the local variable area change. To ensure addressability, a frame pointer must be established to address the local variable area (and register save areas) consistently throughout the function’s activation.

Dynamic stack space allocation requires the following steps.

1. After the stack frame has been acquired at function entry and before the first dynamic stack space allocation, a register is set to the value of the stack pointer. This register is the frame pointer and is used to reference data in the local variable area. The function tag word for the function shall specify the register used as the frame pointer.
2. The size of the dynamic space to be allocated is rounded up to the next 16 byte multiple to maintain the stack’s 16 byte alignment.

3. The stack pointer is decreased by the amount determined in step 2 and then the back chain value is copied to its new location.

The following code examples demonstrate allocating dynamic stack space. The first example allocates a 256 byte space and the second example allocates a 40,000 byte space. A “Store Word with Update” instruction is used to allocate the stack space. This ensures that the stack pointer chain is always preserved.

Figure 7-15: Dynamic Stack Space Allocation Sample Code (Small)

```
ori   %r30, %r1, 0      # r30=frame pointer
...
lwz   %r11, 0(%r1)      # r11=back chain pointer
stwu  %r11, -256(%r1)   # allocate stack and set back chain
```

Figure 7-16: Dynamic Stack Space Allocation Sample Code (Large)

```
ori   %r30, %r1, 0      # r30=frame pointer
...
lwz   %r11, 0(%r1)      # r11=back chain pointer
addis %r12, 0, -1       # set r12 to the complement of
addi  %r12, %r12, 65536-40000 # the allocation size (40000)
stwux %r11, %r1, %r12   # allocate stack and set back chain
```

The process of dynamically allocating stack space can be repeated as many times as necessary within a function’s activation. When the function returns, the stack pointer is set to the back chain value which releases all of the dynamically allocated stack space along with the rest of the stack frame. Naturally, dynamically allocated stack space must not be referenced after the stack frame has been released.

Even in the presence of exceptions or signals, the dynamic allocation process is safe. If dynamic stack space allocation is interrupted, one of three thing can happen.

- The exception or signal handler can return. The dynamic allocation is resumed from the point of interruption.
- The exception or signal handler can execute a non-local goto, or `longjmp`. The thread context is changed to a previous stack frame, automatically discarding the current stack frame including the dynamic allocation.
- The process (or thread) can terminate.

Regardless of when the interruption occurs during the dynamic stack space allocation, the result is a consistent, though possibly dead, process.

8 Resource File Format

This chapter describes the format of resource files and resource collections. Resources are read-only data with the following usage restrictions.

- Resources must be attached to valid load modules.
- Resources can only be accessed via operating system-specific APIs.
- Resources are only mapped into memory by the operating system-specific APIs.
- Resource data may have no relocations and no code may have relocations that refer to the resource data.

From the application view, resources are high-level language-independent, structured data that are used by applications, but are understood by the operating system. *E.g.* fonts, bitmaps, icons, dialogs. See OS/2 for an example of an operating system that uses resources.

8.1 Resource File

A resource file is a load module format-independent file created by a resource compiler. It can contain many uniquely named resource collections and can be converted in to a form suitable for attachment to a load module. This is typically an object file with one `SHT_RES` section per resource collection. The static linker will take this object file as input, along with the other object files that contribute to the load module, and place each resource collection in the resultant load module as a `PT_RES` segment.

8.1.1 Resource File Header

A resource file begins with a resource file header.

Figure 8-1: Resource File Header, `Res32_File`

```
#define RF_NIDENT 16

typedef struct
{
   unsigned char  rf_ident[RF_NIDENT];
   Elf32_Word     rf_rfsize;
   Elf32_Word     rf_rcnum;
   Elf32_Off      rf_rcoff;
} Res32_File;
```

Member	Description
`rf_ident`	The initial bytes which identify this as a resource file and provide machine-independent data with which to decode and interpret the contents. See § 8.1.2, “Resource File Identification”, on page 79 for details.
`rf_rfsize`	This member holds the resource file header’s size in bytes.
`rf_rcnum`	This member holds the number of resource collections in this resource file.
`rf_rcoff`	This member holds the offset of the resource collection array. The offset is relative to the start of the resource file and shall be word aligned. There are `rf_rcnum` entries in the array.

Each entry in the resource collection array is an `Res32_Col` structure. This structure contains the offset and size of a resource collection.

Figure 8-2: Resource Collection Array Entry, `Res32_Col`

```
typedef struct
{
   Elf32_Off      rc_collection;
   Elf32_Word     rc_size;
} Res32_Col;
```

Figure 8-2: Resource Collection Array Entry, `Res32_Col` (Continued)

Member	Description
`rc_collection`	This member holds the offset of a resource collection. The offset is relative to the start of the resource file and shall be word aligned. See § 8.2, “Resource Collection”, on page 81 for details on resource collections.
`rc_size`	This member holds the total size of the resource collection in bytes.

Each resource collection must be self contained and occupy only the space in the resource file beginning at offset `rc_collection` having a length of `rc_size` bytes. All offsets within a resource collection must be relative to the start of the resource collection.

8.1.2 Resource File Identification

Following is a description of the `rf_ident` member of the resource file header.

Table 8-1: `rf_ident[]` Identification Indexes

Name	Value	Description
RF_MAG0 RF_MAG1 RF_MAG2 RF_MAG3	0 1 2 3	0x02 ‘R’ ‘E’ ‘S’ The first 4 bytes hold a “magic number” identifying this as a resource file.
RF_CLASS	4	Class of the data in this resource file.

Name	Value	Meaning
RESCLASSNONE	0	Invalid class
RESCLASS32	1	32 bit architecture. This is the format of the data specified herein.
RESCLASS64	2	This is reserved for 64 bit architectures. It is otherwise unspecified.

Table 8-1: `rf_ident[]` Identification Indexes (Continued)

Name	Value	Description
RF_DATA	5	Data encoding of the data in this resource file. Name / Value / Meaning RESDATANONE / 0 / Invalid data RESDATA2LSB / 1 / Little Endian data encoding RESDATA2MSB / 2 / Big Endian data encoding
RF_VERSION	6	Resource file version. Name / Value / Meaning RV_NONE / 0 / Invalid version RV_CURRENT / 1 / Current version. The value of RV_CURRENT will change as necessary to reflect the current version.
RF_PAD	7	Start of padding bytes. These bytes are reserved and set to zero. The value of RF_PAD will change if currently unused bytes take on meaning.

8.1.3 Resource File PowerPC Processor-specific Information

This section documents the Resource File information specific to the PowerPC processor.

The `rf_ident` member shall contain the following identification values.

Table 8-2: Resource File Identification, `rf_ident`

Position	Value	Meaning
rf_ident[RF_CLASS]	RESCLASS32	32-bit implementation
rf_ident[RF_DATA]	RESDATA2LSB	Little Endian implementation

8.2 Resource Collection

A resource collection is a collection of individual resource items each identified by a unique type and ordinal pair with an optional name (unique within type).

Resource collections are self-contained. All information about the individual resource items are contained within the resource collection. All offsets within a resource collection must be relative to the start of the resource collection. A resource collection includes the resource collection header, the locale information, the resource item array, the resource collection string table and all the resource data.

8.2.1 Resource Header

A resource collection begins with a resource header.

Figure 8-3: Resource Header, `Res32_Hdr`

```
typedef struct
{
   Elf32_Half     rh_version;
   Elf32_Half     rh_flags;
   Elf32_Off      rh_name;
   Elf32_Off      rh_rioff;
   Elf32_Word     rh_rientsize;
   Elf32_Word     rh_rinum;
   Elf32_Word     rh_rhsize;
   Elf32_Off      rh_strtab;
   Elf32_Off      rh_locale;
} Res32_Hdr;
```

Member	Description
`rh_version`	This member holds the resource collection version. The current version is `1`.
`rh_flags`	This member holds the resource collection flags. There are currently no flags defined.
`rh_name`	This member specifies the name of the resource collection. Its value is an index into the resource string table. See `rh_strtab` below.
`rh_rioff`	This member holds the offset of the resource item array. The offset is relative to the start of the resource collection and shall be word aligned. There are `rh_rinum` entries in the array. See § 8.3, “Resource Item”, on page 83 for details on resource items.
`rh_rientsize`	This member holds the size, in bytes, of a resource item array entry.
`rh_rinum`	This member holds the number of resources in the collection.

Figure 8-3: Resource Header, `Res32_Hdr` (Continued)

`rh_rhsize`	This member holds the resource header's size in bytes.
`rh_strtab`	This member holds the offset of the resource string table. The offset is relative to the start of the resource collection. The format of the string table is identical to that of a standard ELF string table. The resource string table shall be located at the end of the resource collection.
`rh_locale`	This member holds the offset of the locale information. The offset is relative to the start of the resource collection and shall be word aligned. The locale information, if present, shall immediately follow the resource header. The locale information is detailed below.

A resource collection may contain resources specific to certain locale. In this case the `rh_locale` member of the resource collection header will hold the offset of the information identifying the locale to which the resource collection applies. If the resource collection is not locale specific, then the `rh_locale` member holds zero indicating no specific locale. The resource collection locale information is detailed below.

Figure 8-4: Resource Collection Locale Information, `Res32_Locale`

```
typedef struct
{
   Elf32_Half     rl_country[2];
   Elf32_Half     rl_language[2];
} Res32_Locale;
```

Member	Description
`rl_country`	This member holds the two character UNICODE representation of the country.
`rl_language`	This member holds the two character UNICODE representation of the language.

8.3 Resource Item

A resource collection contains an array of resource items, with one resource item for each resource in the collection. A resource item describes the resource and provides a pointer to it's data.

Figure 8-5: Resource Item, `Res32_Item`

```
typedef struct
{
   Elf32_Word     ri_type;
   Elf32_Word     ri_ordinal;
   Elf32_Off      ri_name;
   Elf32_Off      ri_data;
   Elf32_Word     ri_size;
} Res32_Item;
```

Member	Description
`ri_type`	This member holds the type of the resource. Its value is operating system-specific. The resource type defines the meaning of the resource data.
`ri_ordinal`	This member holds the ordinal number of the resource. Each resource has a unique ordinal within its resource type.
`ri_name`	This member specifies the name of the resource ordinal. Its value is an index into the resource string table. Names are optional for resources. If the resource has no name, this member holds the value zero.
`ri_data`	This member holds the offset of the resource data. The offset is relative to the start of the resource collection and shall be word aligned. The format of the resource data is operating system-specific.
`ri_size`	This member holds the size of the resource data in bytes.

The resource items shall be sorted by `ri_type` in ascending order. Within each `ri_type` they shall be further sorted by `ri_ordinal` in ascending order. This allows for a resource (type, ordinal) pair to be located via binary search.

9 Object and Load Module File Format

This chapter describes the formats of the object files and load module files (including executables and dynamic link libraries) and debugging information for operating systems and development tools conforming to this ABI.

An *object file* (`ET_REL`) is the output of a language translator that is suitable for linking with other object files to create a load module. A *load module* is either an executable or a dynamic link library (also known as a shared library) and is suitable for loading into a process for execution. An *executable* (`ET_EXEC`) is a load module that is the basis for the creation of a process. It can be dynamically linked with zero or more *dynamic link libraries* (`ET_DYN`) by the system to complete the process image.

9.1 ELF

The format of the object and load module files is based upon the ELF specification, version 1. The reader is referred to the *Tool Interface Standards Portable Formats Specification* and *Executable and Linking Format (ELF)* for a general specification of the ELF format. (See also chapters 4 and 5 of *System V Application Binary Interface*.) Book I of *Executable and Linking Format (ELF)* along with the information in this section describe ELF as required to support this ABI.

9.1.1 ELF Operating System-specific Information

This section documents the non-processor-specific ELF information required for operating systems based upon this ABI. For PowerPC processor-specific information, see § 9.1.2, “ELF PowerPC Processor-specific Information”, on page 104.

9.1.1.1 Sections

The following section types are defined.

Table 9-1: Section Types, `sh_type`

Name	Value	Meaning
`SHT_SYMTAB` `SHT_DYNSYM`	2 11	These sections hold a symbol table. See “Symbol Table” in *Executable and Linking Format (ELF)*, Book I for details. A file may have only one section of each type. Typically, `SHT_SYMTAB` provides symbols for static linking. As a complete symbol table, it may be used for dynamic linking even though it may contain many symbols unnecessary for dynamic linking. A load module may also contain a `SHT_DYNSYM` symbol table which holds a minimal set of symbols necessary for dynamic linking.
`SHT_STRTAB`	3	This section holds a string table. See “String Table” in *Executable and Linking Format (ELF)*, Book I for details. A file may have multiple string tables.
`SHT_OS`	`0x60000001`	This section holds the information to identify the target operating system environment. See § 9.1.1.3, “Operating System Information”, on page 91 for details.
`SHT_IMPORTS`	`0x60000002`	This section holds information on references to external symbols. See § 9.1.1.4, “Import Table”, on page 92 for details.
`SHT_EXPORTS`	`0x60000003`	This section holds information on symbols that are being exported. See § 9.1.1.5, “Export Table”, on page 94 for details.

Table 9-1: Section Types, `sh_type` (Continued)

Name	Value	Meaning
SHT_RES	0x60000004	This section holds read-only resource data. See § 9.1.1.6, “Resource Collection”, on page 95 for details.

The following section attribute flags are defined.

Table 9-2: Section Attribute Flags, `sh_flags`

Name	Value	Meaning
SHF_BEGIN	0x01000000	This section shall be placed at the beginning of the combination of like-named sections during the static link step. Only one of the like-named sections in the combination may have the SHF_BEGIN flag. A section may not have both the SHF_BEGIN and SHF_END flag.
SHF_END	0x02000000	This section shall be placed at the end of the combination of like-named sections during the static link step. Only one of the like-named sections in the combination may have the SHF_END flag. A section may not have both the SHF_BEGIN and SHF_END flag.

These flags are used to control the combination by the static linker of sections having the same name when it may be necessary to have a specific section from one object file placed at the beginning or end of the combined section in the load module.

Two members in the section header, `sh_link` and `sh_info`, hold special information depending on the section type.

Table 9-3: `sh_link` and `sh_info` Interpretation

sh_type	sh_link	sh_info
SHT_SYMTAB SHT_DYNSYM	The section header index of the string table used by entries in this section.	The symbol table index of the last local symbol (binding STB_LOCAL) + 1.
SHT_OS SHT_RES	SHN_UNDEF	0
SHT_IMPORTS	The section header index of the string table used by entries in this section.	0
SHT_EXPORTS	The section header index of the symbol table used by entries in this section.	The section header index of the string table used by entries in this section.

All these sections must be word aligned. Thus the `sh_addralign` value for all these sections is 4.

9.1.1.1.1 Special Sections

The following sections are used by the system and have the indicated types and attributes. They are described below.

Table 9-4: Special Sections

Name	Type	Attributes
`.dynstr`	`SHT_STRTAB`	`SHF_ALLOC`
`.dynsym`	`SHT_DYNSYM`	`SHF_ALLOC`
`.rela`*name*	`SHT_RELA`	see below
`.osinfo`	`SHT_OS`	none
`.imports`	`SHT_IMPORTS`	`SHF_ALLOC`
`.exports`	`SHT_EXPORTS`	`SHF_ALLOC`
`.res`	`SHT_RES`	none

Table 9-5: Special Section Descriptions

Name	Descriptions
`.dynstr`	This section holds strings used in dynamic linking.
`.dynsym`	This section holds symbols used in dynamic linking.
`.rela`*name*	These sections hold relocation table entries. See § 9.1.2.3, “Relocation”, on page 106 for more information. In object files, these sections have no attribute flags. In load modules, these sections have the `SHF_ALLOC` attribute flag if the section relocates information in a `PT_LOAD` segment; otherwise the section has no attribute flags. Conventionally, *`name`* is supplied by the section to which the relocations apply. Thus a relocation section for `.text` normally would have the name `.rela.text`.
`.osinfo`	This section holds the information to identify the target operating system environment. See § 9.1.1.3, “Operating System Information”, on page 91 for details.
`.imports`	The default name of the import table section. This section holds information on references to external symbols. See § 9.1.1.4, “Import Table”, on page 92 for details.

Table 9-5: Special Section Descriptions (Continued)

Name	Descriptions
.exports	The default name of the export table section. This section holds information on symbols that are being exported. See § 9.1.1.5, “Export Table”, on page 94 for details.
.res	The default name of the resource section. This section holds read-only resource data. See § 9.1.1.6, “Resource Collection”, on page 95 for details.

9.1.1.2 Symbol Table

The following symbol bindings are defined.

Table 9-6: Symbol Binding, ELF32_ST_BIND

Name	Value	Meaning
STB_ENTRY	12	This symbol defines an entry point for the load module. This symbol binding is only found in object files and provides a default entry point to the static linker. Only one symbol with this binding may be encountered during static linking. Otherwise treated the same as STB_GLOBAL.

The following symbol types are defined.

Table 9-7: Symbol Type, ELF32_ST_TYPE

Name	Value	Meaning
STT_IMPORT	11	This symbol is a reference to a symbol in another load module (*i.e.* this symbol is imported). The st_value member holds the offset of the import table entry for this symbol. Otherwise treated the same as STT_FUNC.

9.1.1.2.1 Symbol Values

References to symbols defined outside of a load module are resolved using one of two techniques. How a load module has been built determines which of these techniques must be used for all references to symbols defined in other load modules.

1. If the load module’s Dynamic segment has a DT_IMPORT entry, then all references to symbols defined in other load modules will be through the load module’s import table. The symbol table entry referencing a symbol in another load module will have a type of STT_IMPORT and the st_shndx member will contain the section header table index of the import table section. The st_value member will hold the virtual address of the import table entry for the symbol. The import table entry will contain information about the load module where the symbol is defined. See § 9.1.1.4, “Import Table”, on page 92 for details on the import table.

2. If the load module's Dynamic segment does not have a DT_IMPORT entry, then the load module does not have a import table. The st_shndx member of the symbol table entry referencing a symbol in another load module will contain SHN_UNDEF indicating that the symbol is defined in another load module. To locate where the symbol is defined, the load modules specified by the DT_NEEDED entries in the Dynamic segment must be searched. They are searched in the order specified by the ordering of the DT_NEEDED entries. Only those load modules specified by the DT_NEEDED entries are searched. If a Procedure Linkage Table entry exists in the load module for the symbol, the symbol table entry's st_value member shall contain the virtual address of the PLT entry. Otherwise the st_value member shall contain zero.

System V ABI Note: In *System V Application Binary Interface*, a breadth-first search of all the DT_NEEDED entries, beginning with the executable, is performed to locate the symbol. See "Shared Object Dependencies", *System V Application Binary Interface*, for details. This ABI differs from *System V Application Binary Interface* by only searching the DT_NEEDED entries specified by the load module rather than the entire DT_NEEDED tree of the process.

When searching a load module to resolve a symbol reference from another load module, the symbol can be located in one of two ways. How a load module has been built determines which of these techniques must be used when searching a load module for a symbol.

1. If the load module's Dynamic segment has a DT_EXPORT entry, then all symbols available to resolve references from other load modules will be located through the load module's export table. If the reference is from an import table entry with an ordinal number, then a binary search can be performed on the export table entries to locate the export table entry with a matching ordinal number. Otherwise if the reference is from an import table entry with a name or a symbol table entry whose st_shndx member contains SHN_UNDEF, the export table entries can be searched to locate the export table entry with a matching name. See § 9.1.1.5, "Export Table", on page 94 for details on the export table.
2. If the load module's Dynamic segment does not have a DT_EXPORT entry, then the load module does not have an export table. All symbols available to resolve references from other load modules will be located in the load module's dynamic symbol table (DT_SYMTAB). References from an import table entry with only an ordinal number and no name cannot be resolved since there is no ordinal information in the dynamic symbol table. If the reference is from an import table entry with a name or a symbol table entry whose st_shndx member contains SHN_UNDEF, the dynamic symbol table can be searched, via the symbol hash table (DT_HASH), to locate the symbol table entry with a matching name.

Note: All OS/2 load modules (EOS_OS2) shall be built with import and export tables to perform inter-load module symbol resolution.

9.1.1.3 Operating System Information

Multiple operating system environments will be using the ELF format for their objects and load modules. In order to identify the target operating environment for these files, an operating system information section is used to contain this information.

All object and load module files shall contain operating system information. The `Elf32_Os` structure must be present and may be followed by additional operating system-specific information. The presence and structure of the operating system-specific information is defined by the target operating environment.

Figure 9-1: Operating System Information, `Elf32_Os`

```
typedef struct
{
   Elf32_Word     os_type;
   Elf32_Word     os_size;
} Elf32_Os;
```

Member	Description
`os_type`	This member identifies the target operating system type of the file. See Table 9-8, “Operating System Types, os_type”, below for valid types.
`os_size`	Size of the operating system-specific information immediately following this structure. This field shall be set to 0 if no operating system-specific information follows.

The following table defines possible values for `os_type` and their meanings.

Table 9-8: Operating System Types, `os_type`

Name	Value	Meaning
`EOS_NONE`	0	Bad or unknown operating system
`EOS_PN`	1	IBM Microkernel personality neutral
`EOS_SVR4`	2	UNIX System V Release 4 operating system environment
`EOS_AIX`	3	IBM AIX operating system environment
`EOS_OS2`	4	IBM OS/2 operating system, 32 bit environment

9.1.1.3.1 OS/2-specific Information

The `Elf32_OS2Info` structure shall follow the `Elf32_Os` structure in the `PT_OS` segment for all OS/2 load modules (`os_type` equal to `EOS_OS2`). OS/2 object files shall only contain the `Elf32_Os` structure in the `SHT_OS` section.

Figure 9-2: OS/2 Information, `Elf32_OS2Info`

```
typedef struct
{
   unsigned char  os2_sessiontype;
   unsigned char  os2_sessionflags;
   unsigned char  os2_reserved[14];
} Elf32_OS2Info;
```

Member	Description
`os2_sessiontype`	This member holds the OS/2 session type for the load module. See Table 9-9, “OS/2 Session Types, os2_sessiontype”, below for valid types.
`os2_sessionflags`	This member holds the OS/2 session flags for the load module. There are no flags currently defined, so this member shall be set to zero.
`os2_reserved`	This member is reserved and shall be set to zero.

The following table defines possible values for `os2_sessiontype` and their meanings.

Table 9-9: OS/2 Session Types, `os2_sessiontype`

Name	Value	Meaning
`OS2_SES_NONE`	0	No session type. This value is the only valid value for dynamic link libraries. This value is invalid for executables.
`OS2_SES_FS`	1	Full Screen session.
`OS2_SES_PM`	2	Presentation Manager session.
`OS2_SES_VIO`	3	Windowed (character-mode) session.

9.1.1.4 Import Table

An import table holds information on references to external entry points that must be resolved when a load module is dynamically linked into a process image. These entry points are symbols exported by other load modules. An exported entry point can be imported by name or by ordinal. Import by ordinal is preferred since the ordinal can be quickly located in the export table of the entry point exporter. Import by name involves searching for the name of the entry point in the export table of the entry point exporter.

An import table is an array of import table entries. The entries in the table have no sorting requirements. All import tables entries are referenced by symbol table entries. Import tables

can be present in object files and provide information about external entry points to the static linker. This information can also be explicitly provided to the static linker via linker directives. The static linker will use this information to build the import table in the load module for use by the dynamic linker.

Following is the format of an import table entry. The ordinal values 0 and -1 are reserved.

Figure 9-3: Import Table Entry, `Elf32_Import`

```
typedef struct
{
   Elf32_Word     imp_ordinal;
   Elf32_Word     imp_name;
   Elf32_Word     imp_info;
   Elf32_Word     imp_reserved;
} Elf32_Import;
```

Member	Description
`imp_ordinal`	This member identifies the ordinal number of the exported entry point in the providing load module. Its value may be -1 if the exported entry point is referenced by `imp_name`. Reference by ordinal takes precedence over reference by name.
`imp_name`	This member holds a string table index of the exported entry point name. Its value may be zero if the exported entry point is referenced by `imp_ordinal`.
`imp_info`	This member specifies the load module providing the exported entry point. It is interpreted as shown below in Figure 9-4, “imp_info Description”.
`imp_reserved`	This member is reserved and set to zero.

The `imp_info` member holds an index to the name of the load module and describes how to interpret the index. The following code shows how to manipulate the value.

Figure 9-4: `imp_info` **Description**

```
#define ELF32_IMP_TYPE(i) ((i) >> 24)
#define ELF32_IMP_DLL(i) ((i) & 0x00ffffff)
#define ELF32_IMP_INFO(t,d) (((t) << 24) | ((d) & 0x00ffffff))
```

ELF32_IMP_TYPE		ELF32_IMP_DLL
Name	**Value**	**Meaning**
`IMP_IGNORED`	0	This import table entry shall be ignored.
`IMP_STR_IDX`	1	The value is a string table index to the name of the load module. This type is used by import tables in object files.

Figure 9-4: `imp_info` **Description (Continued)**

`IMP_DT_IDX`	2	The value is a reference to a `DT_NEEDED` entry in the Dynamic segment. A value of 1 here refers to the load module referenced by the first `DT_NEEDED` element, a value of 2 refers to the second and so forth. The order of the `DT_NEEDED` elements is therefore significant. A value of 0 is invalid. This type is used by import tables in load modules.

9.1.1.5 Export Table

An export table holds information on exported symbols. Symbols can be exported by name, by ordinal or by both name and ordinal. Exported symbols have an export table entry associated with them.

Export tables can be present in object files and provide information to the static linker about symbols to be exported by the load module. This information can also be explicitly provided to the static linker via linker directives. The static linker will use this information to build the export table in the load module for use by the dynamic linker.

An export table is an array of export table entries. The export table entries are sorted in ascending order by ordinal number to facilitate binary searches on the table. Following is the format of an export table entry.

Figure 9-5: Export Table Entry, `Elf32_Export`

```
typedef struct
{
   Elf32_Word     exp_ordinal;
   Elf32_Word     exp_symbol;
   Elf32_Word     exp_name;
   Elf32_Word     exp_reserved;
} Elf32_Export;
```

Member	Description
`exp_ordinal`	This member holds the exported ordinal number. Its value may be -1 if the symbol is exported only by name.
`exp_symbol`	This member holds the symbol table index of the symbol being exported.
`exp_name`	This member holds the string table index of the exported name. Its value may be zero if the symbol is exported only by ordinal.
`exp_reserved`	This member is reserved and set to zero.

Note: An executable is not permitted to export symbols. Only dynamic link libraries may export symbols.

9.1.1.6 Resource Collection

A resource section (SHT_RES) or resource segment (PT_RES) contains a resource collection of the same class and data encoding as the ELF file containing the section or segment. See § 8, “Resource File Format”, on page 77 for general information on resources and § 8.2, “Resource Collection”, on page 81 for the format of a resource collection.

9.1.1.7 Segments

When the static linker creates the program header table for a load module, it must ensure that the virtual address ranges assigned to PT_LOAD segments are disjoint and are not directly adjacent, *i.e.* there must be one or more bytes of unassigned virtual address space between neighboring segments.

The following segment types are defined.

Table 9-10: Segment Types, p_type

Name	Value	Meaning
PT_OS	0x60000001	This segment holds the information to identify the target operating system environment. See § 9.1.1.3, “Operating System Information”, on page 91 for details. This segment is required in all load modules.
PT_RES	0x60000002	This segment holds read-only resource data. See § 9.1.1.6, “Resource Collection”, on page 95 for details. For program table header entries of this type, p_vaddr shall be zero.

9.1.1.7.1 Segment Permissions

The following segment permission flags are defined.

Table 9-11: Segment Permission Flags, p_flags

Name	Value	Meaning
PF_X	0x1	This segment has execute permission.
PF_W	0x2	This segment has write permission.
PF_R	0x4	This segment has read permission.
PF_S	0x01000000	This segment is shared. The data in this segment is to be shared among all processes using this load module.

Table 9-11: Segment Permission Flags, `p_flags` (Continued)

Name	Value	Meaning
PF_K	0x02000000	This segment is mapped into the kernel address space. This flag is used to indicate that the dynamic link library is a kernel extension and executes at supervisor privilege. Kernel extensions are defined to be memory resident and need not specify the PF_M flag. **Note:** This flag is valid only for dynamic link libraries. If present, this flag must be specified for all PT_LOAD segments in the load module.
PF_M	0x04000000	This segment is memory resident. This flag is used to indicate that the load module is to be memory resident. **Note:** If present, this flag must be specified for all PT_LOAD segments in the load module.
PF_MASKPROC	0xf0000000	There are no processor-specific flags for PowerPC.

If a permission bit is zero, that type of access is denied. Typical flag combinations appear below.

Table 9-12: Typical Segment Permission Flag Combinations

Flags	Meaning
PF_R \| PF_X	Execute/Read (Text segment)
PF_R \| PF_W	Read/Write (Data segment)
PF_R \| PF_W \| PF_S	Read/Write/Shared (Shared data segment)

Note: A typical load module normally has two or three PT_LOAD segments. There is generally always a text segment (PF_R|PF_X) and a data segment (PF_R|PF_W). There may also be a shared data segment (PF_R|PF_W|PF_S). In general, all of the code and data for a load module can be allocated into one of these three segments.

9.1.1.7.2 Segment Contents

The following table illustrates segment contents and the recommended segment ordering for load modules. Some segment types are optional.

Table 9-13: Segment Ordering

Segment	Sections
`PT_OS` Operating system information	`.osinfo`
`PT_INTERP` Program interpreter	`.interp`
`PT_PHDR` Program header table	
`PT_LOAD` (read/execute) Text segment **Note:** Only one executable `PT_LOAD` segment is allowed in a load module.	`.rodata` `.tags` `.imports` `.exports` `.dynamic` (`PT_DYNAMIC` segment) `.dynsym` `.dynstr` `.hash` `.rela.rodata` `.rela.got` `.rela.plt` `.rela.data` `.text` `.got` `.plt`
`PT_LOAD` (read/write) Data segment	`.data` `.bss`
`PT_DYNAMIC` Dynamic segment	This program header table entry describes the `.dynamic` section in the Text segment above.
`PT_*` Segment types not otherwise specified	
`PT_RES` Resource collection	`.res`

Table 9-13: Segment Ordering (Continued)

Segment	Sections
Other data This is data that can be stripped from the load module without affecting the validity of the load module.	.comment .symtab .strtab .line .debug .rela.line .rela.debug .shstrtab

- All SHT_NOBITS sections (*e.g.* .plt and .bss) shall be grouped at the end of their respective segments.
- The .got and .plt sections shall be adjacent with the .got (SHT_PROGBITS) section immediately preceding the .plt (SHT_NOBITS) section.
- The presence of segments and sections within segments is variable. The actual order and membership of sections within a segment may alter the example above.
- While, in general, the system is tolerant of deviations from the recommended segment ordering, some tools which operate on load modules may not be (*e.g.* elfstrip which removes debugging information from load modules). It is therefore strongly urged that the recommended segment ordering be used.

Note: ELF specifies the following ordering requirements for entries in the Program Header table.

- If a PT_INTERP entry is present, it must precede all PT_LOAD entries.
- If a PT_PHDR entry is present, it must precede all PT_LOAD entries. Furthermore, the PT_PHDR segment must be part of the memory image of the load module. This means that the area of the load module file containing the Program Header table which is mapped by the PT_PHDR segment must also be mapped by a PT_LOAD segment.
- All PT_LOAD entries must appear in ascending order sorted by p_vaddr.

9.1.1.8 Dynamic Segment

All load modules shall contain a Dynamic Segment referenced by a segment type of `PT_DYNAMIC`. This segment contains an array of `Elf32_Dyn` structures.

Figure 9-6: Dynamic Structure, `Elf32_Dyn`

```
typedef struct
{
   Elf32_Sword    d_tag;
   union
   {
      Elf32_Word  d_val;
      Elf32_Addr  d_ptr;
   } d_un;
} Elf32_Dyn;
```

Member	Description
`d_tag`	This member identifies the array entry. See Table 9-14, “Dynamic Array Tags, d_tag” below and Table 9-24, “Dynamic Array Tags, d_tag,” on page 111 for possible values. The value of `d_tag` controls in interpretation of the union `d_un`.
`d_val`	This member holds an integer value whose meaning is interpreted by the `d_tag` value.
`d_ptr`	This member holds an address whose meaning is interpreted by the `d_tag` value. When interpreting addresses in the Dynamic Segment, the dynamic linker computes actual addresses, based upon the original file value and the memory base address. For consistency, files do *not* contain relocation entries to “correct” addresses in the Dynamic Segment.

The following dynamic array tags are defined. A “mandatory” tag must be present in the Dynamic Segment. An “optional” tag may be present but is not required. An “ignored” tag may be present but is not used. An “invalid” tag may not be present in a valid load module.

Table 9-14: Dynamic Array Tags, `d_tag`

Name	Value	d_un	Executables	Dynamic Link Libraries
`DT_NULL`	0	ignored	mandatory	mandatory
`DT_NEEDED`	1	`d_val`	optional	optional
`DT_HASH`	4	`d_ptr`	mandatory	mandatory
`DT_STRTAB`	5	`d_ptr`	mandatory	mandatory
`DT_SYMTAB`	6	`d_ptr`	mandatory	mandatory

Table 9-14: Dynamic Array Tags, `d_tag` (Continued)

Name	Value	d_un	Executables	Dynamic Link Libraries
DT_STRSZ	10	d_val	mandatory	mandatory
DT_SYMENT	11	d_val	mandatory	mandatory
DT_SONAME	14	d_val	ignored	mandatory
DT_DEBUG	21	d_ptr	optional	optional
DT_EXPORT	0x60000001	d_ptr	invalid	optional
DT_EXPORTSZ	0x60000002	d_val	invalid	optional
DT_EXPENT	0x60000003	d_val	invalid	optional
DT_IMPORT	0x60000004	d_ptr	optional	optional
DT_IMPORTSZ	0x60000005	d_val	optional	optional
DT_IMPENT	0x60000006	d_val	optional	optional
DT_IT	0x60000007	d_val	invalid	mandatory
DT_ITPRTY	0x60000008	d_val	invalid	mandatory
DT_INITTERM	0x60000009	d_ptr	invalid	mandatory
DT_STACKSZ	0x6000000a	d_val	optional	ignored

The dynamic array tags are described below.

Table 9-15: Dynamic Array Tag Descriptions

Name	Meaning
DT_NULL	This entry marks the end of the Dynamic Segment.
DT_NEEDED	This element holds the string table index, into the string table referenced by DT_STRTAB, of a needed dynamic link library. The ordering of DT_NEEDED elements is significant. See § 9.1.1.2.1, “Symbol Values”, on page 89 for additional information.
DT_HASH	This entry holds the address of the symbol hash table described in § 9.1.1.10, “Hash Table”, on page 103. This hash table refers to the symbol table referenced by the DT_SYMTAB entry.
DT_STRTAB	This entry holds the address of the string table used in dynamic linking. Symbol names, library names and other strings reside in this table.
DT_SYMTAB	This entry holds the address of the symbol table used in dynamic linking.

Table 9-15: Dynamic Array Tag Descriptions (Continued)

Name	Meaning
DT_STRSZ	This entry holds the size, in bytes, of the string table referenced by DT_STRTAB.
DT_SYMENT	This entry holds the size, in bytes, of a symbol table entry in the symbol table referenced by DT_SYMTAB.
DT_SONAME	This element holds the string table index, into the string table referenced by DT_STRTAB, of the name of this dynamic link library.
DT_DEBUG	This entry is used for debugging. Its contents are not specified by this ABI.
DT_EXPORT	This entry holds the address of the export table, described in § 9.1.1.5, “Export Table”, on page 94.
DT_EXPORTSZ	This entry holds the total size, in bytes, of the export table. This entry must accompany a DT_EXPORT entry.
DT_EXPENT	This entry holds the size, in bytes, of an export table entry. This entry must accompany a DT_EXPORT entry.
DT_IMPORT	This entry holds the address of the import table, described in § 9.1.1.4, “Import Table”, on page 92.
DT_IMPORTSZ	This entry holds the total size, in bytes, of the import table. This entry must accompany a DT_IMPORT entry.
DT_IMPENT	This entry holds the size, in bytes, of an import table entry. This entry must accompany a DT_IMPORT entry.
DT_IT	This entry holds the initialization and termination types for a dynamic link library. This entry is mandatory if DT_INITTERM is specified. See § 9.1.1.9, “Initialization and Termination Functions”, on page 102 for details.
DT_ITPRTY	This entry holds a priority value indicating the relative priority of the initialization and termination of this dynamic link library to all other dynamic link libraries in a process. This entry is mandatory if DT_INITTERM is specified. See § 9.1.1.9, “Initialization and Termination Functions”, on page 102 for details.
DT_INITTERM	This entry holds the address of the initialization and termination function. This function performs both initialization and termination duties. This entry will hold zero if there is no initialization and termination function. See § 9.1.1.9, “Initialization and Termination Functions”, on page 102 for details.

Table 9-15: Dynamic Array Tag Descriptions (Continued)

Name	Meaning
DT_STACKSZ	This entry holds the requested size of the stack for the executable. (This is the stack for thread 1 in the process.) The system will create the stack with a size greater than or equal to the specified size. If this entry is not present, then the system will create the stack with a default size.

9.1.1.9 Initialization and Termination Functions

The DT_IT entry identifies the type of initialization and termination behavior for an OS/2 dynamic link library. The following code show how to manipulate its value and its meaning.

Figure 9-7: DT_IT Description

```
#define ELF32_IT_INIT(it) ((it) & 0x0f)
#define ELF32_IT_TERM(it) (((it) >> 4) & 0x0f)
#define ELF32_IT_INFO(i,t) (((i) & 0x0f)|(((t) & 0x0f) << 4))
```

ELF32_IT_INIT and ELF32_IT_TERM

Name	Value	Meaning
IT_NONE	0	No initialization or termination. The dynamic link library is not called for initialization or termination.
IT_GLOBAL	1	Global initialization or termination. If a dynamic link library is not in use by any process and a new process causes the dynamic link library to be loaded, the initialization routine is called. If a process causes a dynamic link library to be unloaded and no other process is using the dynamic link library, the termination routine is called.
IT_INSTANCE	2	Process initialization or termination. If a process causes a dynamic link library to be loaded, the initialization routine is called. If a process causes a dynamic link library to be unloaded, the termination routine is called.
IT_THREAD	3	**Note:** This value is currently not supported but is reserved for future use.

The remaining values are reserved. If initialization or termination is indicated then the DT_INITTERM entry must be present and non-zero. The DT_ITPRTY entry holds a priority value that, when compared with the DT_ITPRTY values of the other dynamic link libraries in a process image, gives a relative ordering of the initialization and termination of the dynamic link libraries in the process image. A value of zero is the highest priority with priority decreasing as the value increases. The order in which dynamic link libraries with the same priority value are initialized and terminated is unspecified.

The initialization and termination function, whose address is specified by DT_INITTERM, is called with two parameters as described below.

Figure 9-8: DT_INITTERM Function Prototype

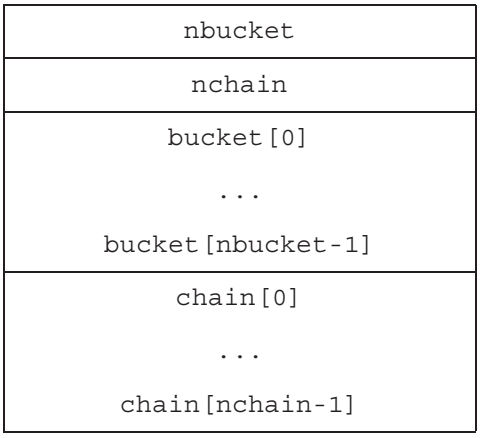
typedef unsigned long INITTERM (unsigned long modhandle, unsigned long flag);		
Parameter	**Description**	
modhandle	This parameter holds a system assigned handle for the load module.	
flag	This parameter holds one of the following values.	
	0	If the entry point is being called for initialization.
	1	If the entry point is being called for termination.
Return Value	A non-zero return value indicates successful initialization/ termination. A zero return value indicates that initialization/ termination failed. If a failure is indicated, the system will abort the load of the dynamic link library. If this failure occurs during process start-up, the system will abort the process.	

See § 11.2, “Process Initialization”, on page 131 and § 11.3, “Process Termination”, on page 133 for more information on dynamic link library initialization and termination.

9.1.1.10 Hash Table

A hash table of Elf32_Word entries supports symbol table access. Labels appear below to help explain the hash table organization, but they are not part of the specification.

Figure 9-9: Symbol Hash Table



The bucket array contains nbucket entries and the chain array contains nchain entries. Both indexes start at zero. Both bucket and chain hold symbol table indexes. Chain table entries parallel the symbol table. The number of symbol table entries should equal nchain; so symbol table indexes also select chain table entries. A hashing function accepts a symbol

name and returns a value that may be used to compute a `bucket` index. Consequently, if the hashing function returns the value *x* for some name, `bucket[`*x*`%nbucket]` gives and index, *y*, into both the symbol table and the chain table. If the symbol table entry is not the one desired, `chain[`*y*`]` gives the next symbol entry with the same hash value. One can follow the `chain` links until either the selected symbol entry holds the desired name or the `chain` entry contains the value `STN_UNDEF`.

Figure 9-10: Hashing Function

```
unsigned long elf_hash ( const unsigned char * name )
{
   unsigned long h = 0, g;

   while ( *name )
   {
      h = ( h << 4 ) + *name++;
      if ( g = h & 0xf0000000 )
         h ^= g >> 24;
      h &= ~g;
   }
}
```

9.1.2 ELF PowerPC Processor-specific Information

This section documents the ELF information specific to the PowerPC processor.

9.1.2.1 ELF Header

9.1.2.1.1 Machine Identification

The `e_ident` member shall contain the following identification values.

Table 9-16: ELF Identification, `e_ident`

Position	Value	Meaning
`e_ident[EI_CLASS]`	`ELFCLASS32`	32-bit implementation
`e_ident[EI_DATA]`	`ELFDATA2LSB`	Little Endian implementation

The `e_machine` member shall contain the following machine type.

Table 9-17: Processor Identification, `e_machine`

Name	Value	Meaning
`EM_PPC`	`20`	PowerPC Architecture

The e_flags member may contain a combination of the following processor-specific flags.

Table 9-18: Processor Flags, e_flags

Name	Value	Meaning
EF_PPC_EMB	0x80000000	Reserved for use in embedded systems.

9.1.2.2 Sections

9.1.2.2.1 Special Sections

The following sections are used by the system and have the indicated types and attributes. They are described below.

Table 9-19: Special Sections

Name	Type	Attributes
.got	SHT_PROGBITS	SHF_ALLOC \| SHF_EXECINSTR
.plt	SHT_NOBITS	SHF_ALLOC \| SHF_EXECINSTR
.tags	SHT_PROGBITS	SHF_ALLOC

Table 9-20: Special Section Descriptions

Name	Descriptions
.got	This section holds the Global Offset Table or GOT. See § 11.5, “Global Offset Table (GOT)”, on page 135 for details.
.plt	This section holds the Procedure Linkage Table or PLT. See § 11.6, “Procedure Linkage Table (PLT)”, on page 137 for details. **Note:** The .plt section on PowerPC is of type SHT_NOBITS, not SHT_PROGBITS as on most other processors.
.tags	This optional section holds the long form function tag information. See § 7.2, “Function Tags”, on page 58 for details on function tags.

System V ABI Note: The *System V Application Binary Interface, PowerPC Processor Supplement* defines both the .got and .plt sections to have the SHF_WRITE attribute. It additionally does not define the .got section to have the SHF_EXECINSTR attribute.

Note: The section names .sdata and .sbss are assigned for use by UNIX System V. Section names beginning with .PPC.EMB. and the section name .rosdata are assigned for use by embedded systems.

System V ABI Note: The *System V Application Binary Interface, PowerPC Processor Supplement* defines a Small Data Area with initialized data in section `.sdata` and uninitialized data in section `.sbss`.

9.1.2.3 Relocation

This ABI specifies the use of only relocation entries with explicit addends, *i.e.* `Elf32_Rela`. Relocations are applied to halfwords or words and the `r_offset` field holds the offset or virtual address of the storage unit to be modified. The `r_addend` member holds the relocation addend. All values use the data encoding specified in the ELF header.

9.1.2.3.1 Relocation Types

The relocation types specify how to compute the relocation value and which bits to modify in the target storage unit.

The following relocation fields are defined. Little Endian byte numbers are in the upper corners. Bit numbers appear in the lower corners.

Table 9-21: Relocation Fields

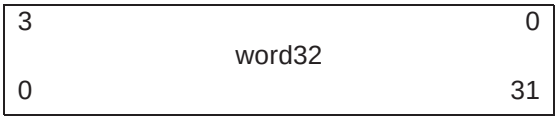
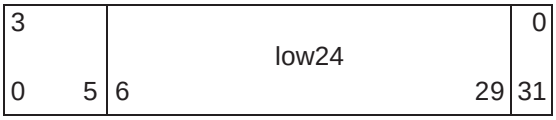
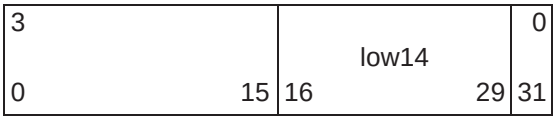
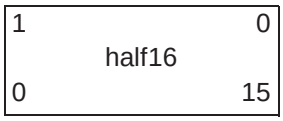
Field	Description
`word32` 	Specifies a 4 byte field with word alignment.
`low24` 	Specifies a 24-bit field contained within an aligned word. The 24-bit relative displacement in bits 6-29 are modified and the remaining bits are ignored and unmodified. (See the LI field of the I Instruction Format in *PowerPC Architecture*.)
`low14` 	Specifies a 14-bit field contained within an aligned word. The 14-bit relative displacement in bits 16-29 are modified and bit 10 (branch prediction) may be modified depending on the relocation type. The remaining bits are ignored and unmodified. (See the BO_4 and BD fields of the B Instruction Format in *PowerPC Architecture*.)

Table 9-21: Relocation Fields (Continued)

Field	Description
`half16` 	Specifies a 2 byte field with halfword alignment. (See the D, SI or UI fields of the D Instruction Format in *PowerPC Architecture*.)

The following variables are defined for use in the relocation calculations.

Table 9-22: Relocation Variables

Variable	Definition
`A`	Addend to be used in the relocation calculation. The `r_addend` member of the relocation entry holds this value.
`B`	Adjustment to the base address of a segment. When the load module is created by the static linker, each segment is created with a base address. This is specified by the `p_vaddr` member of the program header table entry for the segment. The difference between the actual load address for the segment and the address specified in `p_vaddr` is the value of `B`. Each segment will have its own, possibly unique, value of `B`. When using the variable `B` in a relocation calculation (for `R_PPC_RELATIVE`), the value being relocated must be examined to determine which segment it references. The value of `B` used in the calculation will be the value of `B` corresponding to the referenced segment.
`G`	Offset in the Global Offset Table (from the symbol `_GLOBAL_OFFSET_TABLE_`) of the GOT entry which will contain the address of the relocation entry's symbol. See § 11.5, "Global Offset Table (GOT)", on page 135 for more information.
`L`	Section offset or virtual address of the Procedure Linkage Table entry of the relocation entry's symbol. See § 11.6, "Procedure Linkage Table (PLT)", on page 137 for more information.
`P`	Section offset or virtual address of the storage unit being relocated. The `r_offset` member of the relocation entry holds this value.
`S`	Value of the relocation entry's symbol. The `st_value` member of the symbol table entry referenced by the relocation entry holds this value.

Section offsets are used in object files and virtual addresses are used in load modules.

The following general rules apply to the interpretation of the relocation types in Table 9-23, "Relocation Types".

- The value of the calculation replaces the value in the field being relocated. In no case does the value in the field participate in the calculation.
- "+" and "-" denote 32-bit modulus addition and subtraction, respectively, of the left and right operands. ">>" denotes arithmetic right shifting (shift with sign copy) of the left operand by the number of bits specified by the right operand.
- `@l(`*value*`)` returns the least significant 16-bits of the 32-bit *value*. `@h(`*value*`)` returns the most significant 16-bits of the 32-bit *value*. `@ha(`*value*`)` returns the most significant 16-bits of the 32-bit *value* adjusted for `@l(`*value*`)` being treated as a signed number.
 - `@l(`*x*`) = (`*x*` & 0xFFFF)`
 - `@h(`*x*`) = ((`*x*` >> 16) & 0xFFFF)`
 - `@ha(`*x*`) = (((`*x*` >> 16) + ((`*x*` & 0x8000) ? 1 : 0)) & 0xFFFF)`
- References to the variable `G` in the calculation implicitly directs the static linker to create a GOT entry for the referenced symbol.
- References to the variable `L` in the calculation implicitly directs the static linker to create a PLT entry for the referenced symbol.
- The static linker shall report all relocation failures as errors.

Table 9-23: Relocation Types

Name	Value	Field	Calculation	Notes
R_PPC_NONE	0	none	none	5
R_PPC_ADDR32	1	word32	S + A	5
R_PPC_ADDR24	2	low24	(S + A) >> 2	1,2,4
R_PPC_ADDR16	3	half16	S + A	1,3
R_PPC_ADDR16_LO	4	half16	@l(S + A)	
R_PPC_ADDR16_HI	5	half16	@h(S + A)	
R_PPC_ADDR16_HA	6	half16	@ha(S + A)	
R_PPC_ADDR14	7	low14	(S + A) >> 2	1,3,4
R_PPC_ADDR14_BRTAKEN	8	low14	(S + A) >> 2	1,3,4,7
R_PPC_ADDR14_BRNTAKEN	9	low14	(S + A) >> 2	1,3,4,7
R_PPC_REL24	10	low24	(S + A - P) >> 2	1,2,4
R_PPC_REL14	11	low14	(S + A - P) >> 2	1,3,4
R_PPC_REL14_BRTAKEN	12	low14	(S + A - P) >> 2	1,3,4,7
R_PPC_REL14_BRNTAKEN	13	low14	(S + A - P) >> 2	1,3,4,7
R_PPC_GOT16	14	half16	G + A	1,3,6

Table 9-23: Relocation Types (Continued)

Name	Value	Field	Calculation	Notes
R_PPC_GOT16_LO	15	half16	@l(G + A)	6
R_PPC_GOT16_HI	16	half16	@h(G + A)	6
R_PPC_GOT16_HA	17	half16	@ha(G + A)	6
R_PPC_PLTREL24	18	low24	(L + A - P) >> 2	1,2,4,8
R_PPC_COPY	19	none	see note 9	5,9
R_PPC_GLOB_DAT	20	word32	S + A	5,10
R_PPC_JMP_SLOT	21	none	see note 11	5,11
R_PPC_RELATIVE	22	word32	B + A	5,12
R_PPC_LOCAL24PC	23	low24	see note 13	1,2,4,13
R_PPC_UADDR32	24	word32	S + A	14
R_PPC_UADDR16	25	half16	S + A	1,3,14
R_PPC_REL32	26	word32	S + A - P	
R_PPC_PLT32	27	word32	L + A	8
R_PPC_PLTREL32	28	word32	L + A - P	8
R_PPC_PLT16_LO	29	half16	@l(L + A)	8
R_PPC_PLT16_HI	30	half16	@h(L + A)	8
R_PPC_PLT16_HA	31	half16	@ha(L + A)	8
R_PPC_SDAREL16	32	half16	see note 9	1,3,9
R_PPC_SECTOFF	33	half16	see note 9	1,3,9
R_PPC_SECTOFF_LO	34	half16	see note 9	9
R_PPC_SECTOFF_HI	35	half16	see note 9	9
R_PPC_SECTOFF_HA	36	half16	see note 9	9
R_PPC_REL30	37		see note 9	9

The entries in the Notes column are defined below.

1. The relocation is subject to failure if the computed value does not fit within the bits specified by the Field column.
2. The most significant 7 bits of the computed value, before any shifting, must all be the same.

3. The most significant 17 bits of the computed value, before any shifting, must all be the same.
4. The least significant 2 bits of the computed value, before any shifting, must be zero.
5. This relocation type may be seen by the dynamic linker
6. This relocation type directs the static linker to build a Global Offset Table and to add a GOT entry to the table which will contain the address of the relocation entry's referenced symbol. There will be only one GOT entry for each symbol.
7. This relocation type indicates that the branch prediction bit (bit 10) of the conditional branch instruction being relocated should be modified. The `_BRTAKEN` suffix predicts the branch will be taken. The `_BRNTAKEN` suffix predicts the branch will not be taken.
8. This relocation type directs the static linker to build a Procedure Linkage Table and to add a PLT entry to the table for the relocation entry's referenced symbol. There will be only one PLT entry for each symbol. Since the PLT entry is assumed to be ± 32MB from the call site, the `R_PPC_PLTREL24` relocation type is the only type needed to relocate branches to PLT entries.
9. This relocation type is not used in this ABI but is assigned for use by UNIX System V.
10. This relocation type is used to set a Global Offset Table entry to the address of the specified symbol. It is otherwise identical to `R_PPC_ADDR32` but allows a correspondence between symbols and GOT entries to be determined.
11. This relocation type directs the dynamic linker to relocate Procedure Linkage Table entries. The `DT_JMPREL` entry in the Dynamic segment points to an array of these relocation entries. There is a one-to-one correspondence between the relocation entries and the PLT entries. See § 11.6, "Procedure Linkage Table (PLT)", on page 137 for additional details.
12. This relocation type directs the dynamic linker to perform a relative relocation. The `r_addend` member holds a pointer to which the relative relocation is to be applied. The pointer was initialized by the static linker based upon the `p_vaddr` member of the program header table entry for the segment into which the pointer points. The pointer must be relocated based upon the load address of that segment. The pointer is adjusted by the difference between the actual load address and the `p_vaddr` address of the segment into which the pointer points (the relocation variable `B`). The storage unit addressed by `r_offset` is then assigned the adjusted pointer. This relocation type shall specify no symbol, *i.e.* the symbol table index shall be zero.

 The static linker shall also store the pointer to which the relative relocation is to be applied in the storage unit addressed by `r_offset` as well as in the `r_addend` member. This will allow the dynamic linker to avoid processing `R_PPC_RELATIVE` relocations if all the segments are loaded at the virtual addresses specified by their respective `p_vaddr` members since the storage units will already contain the correct value (because the relocation variable `B` will be zero for all segments in the load module).
13. This relocation type uses the value of the symbol from the individual object file rather than the adjusted symbol value after combining all objects. It is otherwise identical to `R_PPC_REL24`. The referenced symbol for this relocation type is normally

_GLOBAL_OFFSET_TABLE_, which additionally directs the static linker to build a Global Offset Table.

14. This relocation type allows the relocated storage unit to be unaligned.

Relocation values in the range 101-200 and names beginning with R_PPC_EMB_ are assigned for embedded system use. All relocation values not listed in Table 9-23, “Relocation Types”, or otherwise assigned are reserved.

System V ABI Note: The *System V Application Binary Interface, PowerPC Processor Supplement* defines the relocation types R_PPC_COPY, R_PPC_SDAREL16 and R_PPC_SECTOFF*. These relocation types are not supported by this ABI.

9.1.2.4 Dynamic Segment

The following processor-specific dynamic array tags are defined. A “mandatory” tag must be present in the Dynamic Segment. An “optional” tag may be present but is not required.

Table 9-24: Dynamic Array Tags, d_tag

Name	Value	d_un	Executables	Dynamic Link Libraries
DT_PLTRELSZ	2	d_val	optional	optional
DT_PPC_PLT DT_PLTGOT	3	d_ptr	optional	optional
DT_RELA	7	d_ptr	mandatory	mandatory
DT_RELASZ	8	d_val	mandatory	mandatory
DT_RELAENT	9	d_val	mandatory	mandatory
DT_PLTREL	20	d_val	optional	optional
DT_JMPREL	23	d_ptr	optional	optional
DT_PPC_GOT	0x70000001	d_ptr	optional	optional
DT_PPC_GOTSZ	0x70000002	d_val	optional	optional
DT_PPC_PLTSZ	0x70000003	d_val	optional	optional

The dynamic array tags are described below.

Table 9-25: Dynamic Array Tag Descriptions

Name	Meaning
DT_PLTRELSZ	This entry holds the total size, in bytes, of the relocation entries associated with the Procedure Linkage Table. This entry must accompany a DT_JMPREL entry.
DT_PPC_PLT DT_PLTGOT	This entry holds the address of the Procedure Linkage Table, *i.e.* the address of the .plt section. For additional information on the Procedure Linkage Table, see § 11.6, “Procedure Linkage Table (PLT)”, on page 137.
DT_RELA	This entry holds the address of the relocation table. This relocation table holds all relocation entries except those for the Procedure Linkage Table. They are contained in the relocation table addressed by DT_JMPREL. The relocation entries in this table shall be sorted by r_offset values in ascending order. This will allow the dynamic linker to easily locate all relocations for given memory page. **Note:** Since only relocation entries with explicit addends are specified in this ABI, all relocation entries are of the form Elf32_Rela.
DT_RELASZ	This entry holds the total size, in bytes, of the relocation table referenced by DT_RELA. This entry must accompany a DT_RELA entry.
DT_RELAENT	This entry holds the size, in bytes, of a relocation table entry in the relocation table referenced by DT_RELA. This entry must accompany a DT_RELA entry.
DT_PLTREL	This entry specifies the type of relocation entries in the DT_JMPREL relocation table. This entry shall hold the value DT_RELA. This entry must accompany a DT_JMPREL entry.
DT_JMPREL	This entry holds the address of the relocation table for the Procedure Linkage Table. This relocation table only holds relocation entries for the Procedure Linkage Table. All other relocation entries are contained in the relocation table addressed by DT_RELA. The relocation entries in this relocation table have a one-to-one correspondence with the Procedure Linkage Table entries. See § 11.6, “Procedure Linkage Table (PLT)”, on page 137 for further details.
	System V ABI Note: The *System V Application Binary Interface, PowerPC Processor Supplement* specifies that the table of DT_JMPREL relocation entries is wholly contained within the relocation table referenced by DT_RELA. This ABI specifies that the DT_JMPREL table is a separate table from the DT_RELA table.

Table 9-25: Dynamic Array Tag Descriptions (Continued)

Name	Meaning
DT_PPC_GOT	This entry holds the address of the Global Offset Table, *i.e.* the address of the .got section. This is not the same at the address of the symbol _GLOBAL_OFFSET_TABLE_ since negative offsets from _GLOBAL_OFFSET_TABLE_ are allowed. For additional information on the Global Offset Table, see § 11.5, “Global Offset Table (GOT)”, on page 135.
DT_PPC_GOTSZ	This entry holds the total size, in bytes, of the Global Offset Table, *i.e.* the size of the .got section. This entry must accompany a DT_PPC_GOT entry.
DT_PPC_PLTSZ	This entry holds the total size, in bytes, of the Procedure Linkage Table, *i.e.* the size of the .plt section. This entry must accompany a DT_PPC_PLT entry.

9.2 DWARF

This section defines the information necessary to support the Debug With Arbitrary Record Format (DWARF) debugging information format on the PowerPC architecture. This ABI does not define a debugging information format, but all implementations of DWARF for this ABI shall use the definitions in this section. The reader is referred to *Tool Interface Standards Portable Formats Specification* for general information on the DWARF debugging information format.

9.2.1 DWARF PowerPC Processor-specific Information

This section documents the DWARF information specific to the PowerPC processor.

9.2.1.1 Register Numbers

The following register number mappings are specified for the PowerPC User Instruction Set Architecture (UISA) registers.

Table 9-26: PowerPC UISA Register Mapping

Name	Abbreviation	Number
General Purpose Registers	`r0` - `r31`	0 - 31
Floating Point Registers	`f0` - `f31`	32 - 63
Condition Register	`CR`	64
Floating Point Status and Control Register	`FPSCR`	65
Fixed Point Exception Register	`XER` (or `SPR1`)	101
Link Register	`LR` (or `SPR8`)	108
Count Register	`CTR` (or `SPR9`)	109
Other Special Purpose Registers	`SPR`*n*	100 + *n*

Note: In general, all registers which have a Special Purpose Register (`SPR`) number will have a DWARF register number equal to 100 plus the SPR number.

The following register number mappings are specified for the PowerPC Virtual Environment Architecture (VEA) registers.

Table 9-27: PowerPC VEA Register Mapping

Name	Abbreviation	Number
Time Base Lower Register (read only)	`TBR 268`	368
Time Base Upper Register (read only)	`TBR 269`	369

The following register number mappings are specified for the PowerPC Operating Environment Architecture (OEA) registers.

Table 9-28: PowerPC OEA Register Mapping

Name	Abbreviation	Number
Machine State Register	`MSR`	66
Segment Registers	`SR0` - `SR15`	70 - 85
Other Special Purpose Registers	`SPR`*n*	100 + *n*

Consult the hardware reference manuals, *PowerPC Architecture* and *PowerPC 603 RISC Microprocessor User's Manual*, for specific Special Purpose Register numbers and uses. The presence of Special Purpose Registers will vary among PowerPC implementations.

10 Object Library File Format

This chapter describes the format of object library files. Object library files are searched by the static linker to resolve undefined symbols. The ABI defines two object library file formats. The Library File format is the preferred file format.

10.1 Archive File Format

This file format is the same as the Archive File format (AR) used in UNIX System V. See *System V Application Binary Interface*, Chapter 7, for details on the Archive File format.

10.2 Library File Format

The Library File format (LIB) is a modification of the AR file format. A LIB file contains only ELF object files.

10.2.1 LIB File Layout

A LIB file has the following layout in the indicated order.

Figure 10-1: LIB File Layout

LIB file header.
An optional special member: symbol table (created only if at least one member defines non-local symbols.)
An optional special member: long file name string table (created only if at least one member's file name exceeds 15 bytes in length.)
An optional special member: full file name string table.
ELF Object File members.

10.2.2 LIB Header

A LIB file begins with a LIB file header. All information in the header is viewable ASCII. All fields have their values stored in ASCII representation and are left-justified and padded on the right with spaces.

Figure 10-2: LIB File Header, `Lib32_File`

```
#define  LIBMAG   "!<mlib>\n"

typedef struct
{
   unsigned char  lib_magic[8];
   unsigned char  lib_class[4];
   unsigned char  lib_data[4];
} Lib32_File;
```

Field	Description
`lib_magic`	The field identifies the file as a LIB file and contains the sequence of characters in `LIBMAG`. **Note:** The sequence of characters is not null-terminated.
`lib_class`	This field holds the ASCII decimal representation of the class of ELF object files in the LIB file. All members of the LIB file shall be of the same class. See the `e_ident[EI_CLASS]` member of the ELF Header for possible values.

Figure 10-2: LIB File Header, `Lib32_File` (Continued)

`lib_data`	This field holds the ASCII decimal representation of the data encoding of ELF object files in the LIB file. All members of the LIB file shall have the same data encoding. See the `e_ident[EI_DATA]` member of the ELF Header for possible values.

Note: For this ABI, the value of `lib_class` shall be `ELFCLASS32` and the value of `lib_data` shall be `ELFDATA2LSB`.

10.2.3 LIB Members

A LIB file member consists of a member header followed by the unchanged contents of the member's file. All member headers begin on a halfword boundary. A newline character ('`\n`') is used if necessary to pad between members. There is no provision for empty areas in the LIB file.

All information in a member header is viewable ASCII. All fields have their values stored in ASCII representation and are left-justified and padded on the right with spaces.

Figure 10-3: LIB Member Header, `Lib32_Hdr`

```
#define  LIBFMAG  "`\n"

typedef struct
{
   unsigned char  lib_name[16];
   unsigned char  lib_date[12];
   unsigned char  lib_uid[6];
   unsigned char  lib_gid[6];
   unsigned char  lib_mode[8];
   unsigned char  lib_size[10];
   unsigned char  lib_fmag[2];
} Lib32_Hdr;
```

Field	Description
`lib_name`	This field contains the member's file name. If the member's file name is 15 bytes or less in length, it is stored in this field, terminated with a slash ('`/`') and padded with spaces on the right (*e.g.* "`filename.o/` "). If the member's file name exceeds 15 bytes in length, the field contains a slash followed by the ASCII decimal representation of the name's offset in the long file name string table, padded with spaces on the right (*e.g.* "`/125` ").
`lib_date`	This field contains the ASCII decimal representation of the number of seconds since 00:00:00 Universal Time, January 1, 1970. For file members, this is the value of the last modification date of the file at the time of its insertion into the library. For special members, the value is zero.

Figure 10-3: LIB Member Header, `Lib32_Hdr` (Continued)

`lib_uid`	This field contains the ASCII decimal representation of the user identifier of the member file at the time of its insertion into the library. For special members, the value is zero. **Note:** The value is zero if the file system does not support user identifiers.
`lib_gid`	This field contains the ASCII decimal representation of the group identifier of the member file at the time of its insertion into the library. For special members, the value is zero. **Note:** The value is zero if the file system does not support group identifiers.
`lib_mode`	This field contains the ASCII octal representation of the member's file access mode at the time of its insertion into the library. For special members, the value is zero. **Note:** The value is zero if the file system does not support access modes.
`lib_size`	This field contains the ASCII decimal representation of the member's size in bytes, exclusive of padding. The member's contents begin immediately following the member header. For special members, the value is the size of the special members data.
`lib_fmag`	The field contains the sequence of characters in `LIBFMAG`. **Note:** The sequences of characters is not null-terminated.

The member's unchanged contents immediately follow the member header.

10.2.4 LIB Special Members

There are three optional special members in a LIB file. If present, they precede all "normal" members in the LIB file and are encountered in the following order.

10.2.4.1 Symbol Table Member

If at least one member of the LIB defines non-local symbols (*i.e.* weak and/or global), the LIB file's first member will be a symbol table member. This member has a special name consisting of a slash followed by spaces ("`/ `").

The data for this member is partitioned as described below. All words in the symbol table are 4 bytes in length and use the data encoding indicated in the `lib_data` member of the file header.

Table 10-1: Symbol Table Layout

	Size	Description
Symbol count	One word.	This word contains the number (`N`) of symbols in the symbol table.
File offset array	`N` words	Each word in the array is an offset, relative to the beginning of the file, to a member of the LIB file. There is a one-to-one correspondence between symbol names in the string table below and entries in this array. The entries in this array provide the offset of the member defining the corresponding symbol.
Symbol information array	`N` bytes	Each byte in the array contains the symbol's type and binding information. It is the equivalent to the `st_info` information found in ELF symbol tables. There is a one-to-one correspondence between symbol names in the string table below and entries in this array. The entries in this array provide the symbol's type and binding information for the corresponding symbol.
String table	`lib_size` - 5*(`N`) - 4	This table consists of a sequence of `N` null-terminated symbol names. Entries in the string table parallel the order of members in the LIB file. Thus if two or more members define symbols with the same name (which is allowed), the string table entries for the symbols will appear in the same order as their corresponding members in the LIB file.

The following example symbol table contains 4 symbols and is 50 bytes in length. The LIB file member at offset 0x114 defines `name`, symbol information `STB_WEAK`, `STT_FUNC`, and `object`, symbol information `STB_GLOBAL`, `STT_OBJECT`. The LIB file member at offset 0x426 defines `function`, symbol information `STB_GLOBAL`, `STT_FUNC`, and `name`, symbol information `STB_GLOBAL`, `STT_FUNC`.

Figure 10-4: Symbol Table Example

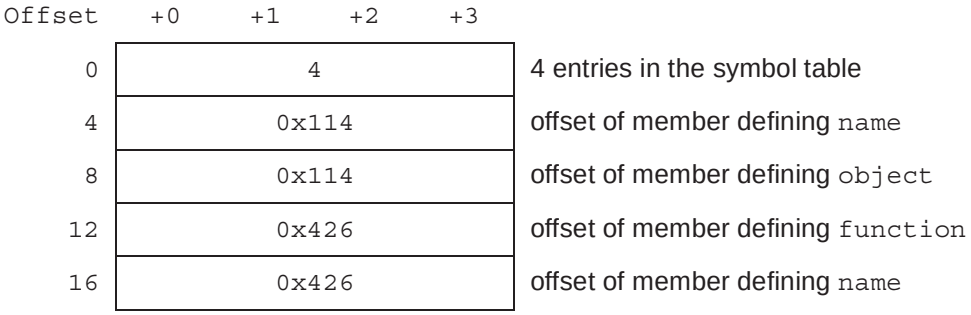


Figure 10-4: Symbol Table Example (Continued)

20	0x22	0x11	0x12	0x12	symbol information
24	n	a	m	e	start of the string table
28	\0	o	b	j	
32	e	c	t	\0	
36	f	u	n	c	
40	t	i	o	n	
44	\0	n	a	m	
48	e	\0			

10.2.4.2 Long File Name String Table Member

If at least one member's file name exceeds 15 bytes in length, a long file name string table member will be present. It follows the symbol table member. This member has a special name consisting of two slashes followed by spaces ("// ").

The data for this member is a sequence of file names, each terminated by a two character sequence of slash and newline ("/\n"). The first byte of the data is offset zero in the long file name string table. The following example shows `lib_name` values for various file names.

Figure 10-5: Long File Name String Table Example

Member Name	lib_name	Comment
regularname.o	regularname.o/	Not in string table
fullbigfilename.o	/0	Offset 0 in string table
evenbiggerfilename.obj	/19	Offset 19 in string table

Offset	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	f	u	l	l	b	i	g	f	i	l
10	e	n	a	m	e	.	o	/	\n	e
20	v	e	n	b	i	g	g	e	r	f
30	i	l	e	n	a	m	e	.	o	b
40	j	/	\n							

10.2.4.3 Full File Name String Table Member

An optional special member may exist which contains the full file names for each member in the LIB file. It follows the symbol table member and the long file name string table member.

This member has a special name consisting of three slashes padded on the right by spaces (“/// “).

The data for this member is a sequence of null-terminated full file names. There is a one-to-one correspondence between entries in the full file name string table and members in the LIB file. If the full file name is not available, then the file name (same as `lib_name`) is used.

Following is an example of a full file name string table.

Figure 10-6: Full File Name String Table Example

Offset	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	/	s	r	c	/	t	e	s	t	/
10	r	e	g	u	l	a	r	n	a	m
20	e	.	o	\0	/	d	e	b	u	g
30	/	f	u	l	l	b	i	g	f	i
40	l	e	n	a	m	e	.	o	\0	e
50	v	e	n	b	i	g	g	e	r	f
60	i	l	e	n	a	m	e	.	o	b
70	j	\0								

11 Process Creation and Dynamic Loading

This chapter contains information about process creation and the initial runtime environment for a newly created process. It also contains information on the Global Offset Table and the Procedure Linkage Table and how they are initialized at process creation.

11.1 Process Virtual Address Space

The 4 GB virtual address space of a process is partitioned into four areas.

Table 11-1: Virtual Address Space Layout

Start Address	Area Name
0x00000000	Private Memory Area
(See below)	Coerced Memory Area
(See below)	Global Memory Area
0xf0000000	System Reserved Area

Area Name	Description
Private Memory Area	This area contains the process' private memory. It contains virtual addresses from 0 up to the start of the Coerced Memory Area.
Coerced Memory Area	This memory area is used for memory objects that are coerced to the same virtual address for all process. When a memory object is allocated in this memory area, the virtual address range of the object is reserved in the virtual address space of every process. However, the memory objects in this area must be explicitly mapped into a process in order to be visible by that process. A memory object can be shared by mapping it into multiple processes. Memory objects in this area can have different access protections for each process mapping the object. The size of the Coerced Memory Area is configurable and is set at boot time.
Global Memory Area	This memory area is used for global memory objects that are coerced to the same virtual address for all process. When a memory object is allocated in this memory area, the virtual address range of the object is reserved in the virtual address space of every process. The object is immediately visible in all processes, no explicit mapping is necessary. Thus memory objects in the Global Memory Area are shared with all processes. All processes have the same access protection to the memory object. The size of the Global Memory Area is configurable and is set at boot time.
System Reserved Area	This memory area is reserved for use by the system. It is 256 MB in size and occupies the virtual address range 0xf0000000 to 0xffffffff. This area is not addressable by the process.

When an executable or dynamic link library is loaded into a process, the various PT_LOAD segments are allocated to specific memory areas. All segments are coerced to the same virtual address in each process that contains the segment image.

Table 11-2: Load Module Memory Area Usage

PT_LOAD Segment	Executable	Dynamic Link Library
PF_R \| PF_X (includes Code, GOT, PLT and read-only data)	Private Memory Area This is shared with other processes running the same executable.	Global Memory Area This is shared with all processes.
PF_R \| PF_W (read/write data)	Private Memory Area This is not shared with other processes but each process' copy resides at the same virtual address.	Coerced Memory Area This is not shared with other processes but each process' copy resides at the same virtual address.
PF_R \| PF_W \| PF_S (shared read/write data)	Private Memory Area This is shared with other processes running the same executable.	Coerced Memory Area This is shared with all other processes using the dynamic link library.

In order to efficiently support demand paging from load modules, the load modules must be statically linked with the file offsets and virtual addresses of segments congruent modulo the page size. Although the current page size for PowerPC is 4096 bytes, file offsets, p_offset, and virtual addresses, p_vaddr, of segments shall be congruent modulo 64K (0x10000) or larger powers of 2. This allows files to be suitable for paging if/when implementations with larger page sizes appear. The value of the p_align member of each program header entry shall be 0x10000.

Following is an example of an executable file that has been linked with a base address of `0x00010000` (64K)

Figure 11-1: Example Executable File

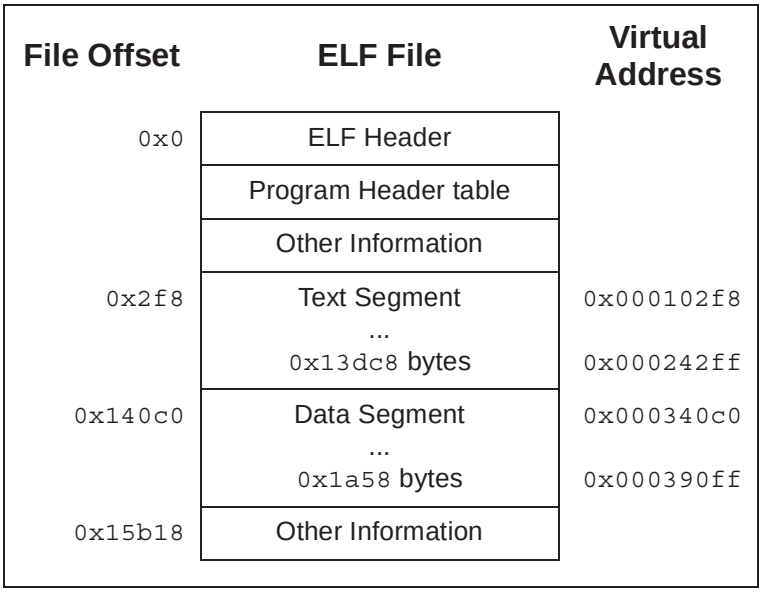


Table 11-3: Example Executable Program Header Table Entries

Member	Text Segment	Data Segment
p_type	PT_LOAD	PT_LOAD
p_offset	0x2f8	0x140c0
p_vaddr	0x000102f8	0x000340c0
p_paddr	0 (unspecified)	0 (unspecified)
p_filesz	0x13dc8	0x1a58
p_memsz	0x14008	0x5040
p_flags	PF_R \| PF_X	PF_R \| PF_W
p_align	0x10000	0x10000

Even though the file offsets and the virtual addresses of the load segments are congruent modulo 64K, up to four pages in the ELF file can hold impure text or data.

- The first page of the text segment may contain the ELF header and other non-text information.
- The last page of the text segment may contain a copy of the beginning of the data segment.
- The first page of the data segment may contain a copy of the end of the text segment.

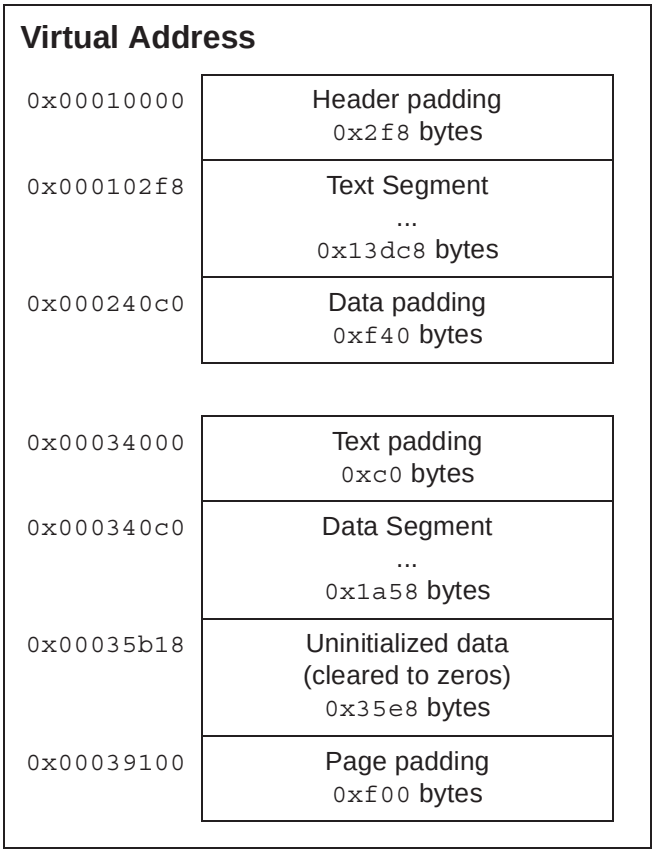
- The last page of the data segment may contain non-data information.

Logically, the system enforces memory access permissions as if each load segment were complete and separate in the ELF file. Segment addresses are adjusted to ensure that each logical page in the virtual address space has a single set of memory access permissions.

In Figure 11-1, “Example Executable File”, above, the file page spanning the end of the text segment and the beginning of the data segment, file offset `0x14000` to `0x15000`, is mapped into memory twice. First at virtual address `0x24000` for the text segment and second at virtual address `0x34000` for the data segment.

If the data segment has `p_memsz` greater than `p_filesz`, then the memory between `p_filesz` and `p_memsz` holds uninitialized data which is defined by the system to begin with zero values. Thus the memory in this range must be set to zero by the loader.

Figure 11-2: Example Executable Process Image Segments



The relative position of segments in a load module, as specified by the virtual addresses in the program header table, need not be maintained by the loader. The loader can locate the segments of a load module at any virtual address without regard to its relative relationship with the other segments in the load module. This is possible for the following reasons.

- All load modules, both executables and dynamic link libraries, contain position independent code.

- The GOT and PLT are located in the text segment and there is only one text segment per load module. All references between the code and the GOT and PLT are intrasegment and thus invariant.
- All intersegment references among data segments shall have a relocations (preferably `R_PPC_RELATIVE`) to adjust them for changes in the relative positions of the segments.

11.2 Process Initialization

This section contains information the initialization of a process. An process consists of an executable and multiple dynamic link libraries dynamically linked together into the process memory image.

An OS/2 dynamic link library is a dynamic link library with an `os_type` of `EOS_OS2`. A Shared Services dynamic link library is a dynamic link library with an `os_type` of `EOS_PN`.

11.2.1 OS/2 Process

The entry point of an OS/2 executable is specified by the `e_entry` member of the ELF header. When control is passed to the entry point, all OS/2 dynamic link libraries in the process have been called for initialization. Any Shared Services dynamic link libraries used by the process must be explicitly called for initialization, if necessary. The contents of the registers at the entry point are as follows.

Table 11-4: OS/2 Process Initial Register State

Register	Contents
`r1`	Stack pointer.
`r2`	Pointer to current thread's Thread Information Block. See § 11.4, "Thread Information Block", on page 134.
`r3`	Module handle for the executable.
`r4`	0 (zero).
`r5`	Pointer to the process' environment data. The environment data is a sequence of null-terminated strings followed by a null byte.
`r6`	Pointer to the command line. The command line is a null-terminated string.
`LR`	Return address to the system.
All other registers	Contents are unspecified.

11.2.1.1 Dynamic Linking of Shared Services Dynamic Link Libraries

Shared Services dynamic link libraries may be dynamically linked into the address space of an OS/2 process. The converse is not true.

11.2.2 Shared Services Process

The entry point of a Shared Services executable is specified by the `e_entry` member of the ELF header. Dynamic link libraries in the process must be explicitly called for initialization, if necessary. The contents of the registers at the entry point are as follows.

Table 11-5: Shared Services Process Initial Register State

Register	Contents
`r1`	Stack pointer.
`r2`	Pointer to current thread's Thread Information Block. See § 11.4, "Thread Information Block", on page 134.
`r3`	Pointer to the argument information. The argument information consists of `argc` followed by the `argv` array followed by the `env` array.
All other registers	Contents are unspecified.

11.2.3 OS/2 Dynamic Link Library Initialization

The initialization/termination entry point of an OS/2 dynamic link library is specified by the `DT_INITTERM` entry in the Dynamic Segment. The default value for `DT_INITTERM` is the address of the symbol `_DLL_InitTerm`. An OS/2 dynamic link library may be called at the initialization/termination entry point when it is made part of the process image. This is determined by the `DT_IT` value. The default value of `DT_IT` is `ELF32_IT_INFO(IT_GLOBAL, IT_GLOBAL)`. Shared Services dynamic link libraries must be explicitly called for initialization, if necessary. The contents of the registers upon entry to the initialization/termination entry point are as follows.

Table 11-6: Dynamic Link Library Initialization Register State

Register	Contents
`r1`	Stack pointer.
`r2`	Pointer to current thread's Thread Information Block. See § 11.4, "Thread Information Block", on page 134.
`r3`	Module handle for the dynamic link library.
`r4`	0 (zero) indicating the initialization/termination entry point is being called for initialization.
`LR`	Return address to the system.
All other registers	Contents are unspecified.

11.3 Process Termination

This section contains information the termination of a process. When a process is terminated, all OS/2 dynamic link libraries that are part of the process image may get called for termination. Shared Services dynamic link libraries must be explicitly called for termination, if necessary.

11.3.1 OS/2 Dynamic Link Library Termination

The initialization/termination entry point of an OS/2 dynamic link library is specified by the `DT_INITTERM` entry in the Dynamic Segment. An OS/2 dynamic link library may be called at the initialization/termination entry point when it is removed from the process image. This is determined by the `DT_IT` value. Shared Services dynamic link libraries must be explicitly called for termination, if necessary. The contents of the registers upon entry to the initialization/termination entry point are as follows.

Table 11-7: OS/2 Dynamic Link Library Termination Register State

Register	Contents
`r1`	Stack pointer.
`r2`	Pointer to current thread's Thread Information Block. See § 11.4, "Thread Information Block", on page 134.
`r3`	Module handle for the dynamic link library.
`r4`	1 (one) indicating the initialization/termination entry point is being called for termination.
`LR`	Return address to the system.
All other registers	Contents are unspecified.

11.4 Thread Information Block

The `r2` register always points to the current thread's Thread Information Block. The `r2` register should never be modified by the application.

Figure 11-3: Thread Information Block, `thread_info_block_t`

```
typedef struct
{
   void *   exception;
   void *   stacklow;
   void *   stackhigh;
   void *   extended;
   uint32   version;
   uint32   threadid;
   void *   sync1;
   void *   sync2;
   uint32   reserved1;
   uint32   reserved2;
} thread_info_block_t;
```

Member	Description
`exception`	This member holds the head of the thread's exception handler chain.
`stacklow`	This member holds the lowest valid address in the thread's stack.
`stackhigh`	This member holds the next address greater than the highest valid address in the thread's stack.
`extended`	The member holds the address of the thread's extended thread information block. The extended thread information block will contain thread package-specific information.
`version`	This member holds the version number of the `thread_info_block_t` data structure.
`threadid`	This member holds the thread's id. The thread id is unique within the process.
`sync1` `sync2`	These members are reserved for synchronization.
`reserved1` `reserved2`	These members are reserved for future use.

Note: `uint32` is an unsigned 32-bit integer data type.

See the OS/2 Programming documentation for a description of the extended thread information block for OS/2 processes.

11.5 Global Offset Table (GOT)

In general, position independent code cannot contain absolute virtual addresses. A load module uses its Global Offset Table to hold the absolute virtual addresses that cannot be held in the code. Using position independent techniques, the code obtains the addresses of memory objects from the GOT and can then access the objects.

The Global Offset Table is initialized, by the static linker, to contain the information needed by its relocation entries (See § 9.1.2.3, “Relocation”, on page 106 for additional information on relocation types). Entries in the GOT are relocated only if they have associated relocation entries, some of which are `R_PPC_RELATIVE` and `R_PPC_GLOB_DAT`. The dynamic linker processes the relocation entries and sets the GOT entries to the resultant values.

Each load module has its own Global Offset Table and consequently a symbol may appear in several tables. The dynamic linker processes all Global Offset Table relocations for all load modules before transferring control to the process. This ensures that all absolute virtual addresses are valid and available for the process code. Once a process begins execution its memory segments must remain at fixed virtual addresses. The Global Offset Table resides in the `.got` section referenced by the `DT_GOT` entry in the Dynamic segment

The symbol `_GLOBAL_OFFSET_TABLE_` is used to access the Global Offset Table and is the reference address for the GOT. The symbol may reside in the center of the GOT allowing for both positive and negative offsets into the array of GOT entries. Four entries in the Global Offset Table are reserved.

Table 11-8: Reserved Global Offset Table Entries

Entry	Description
`_GLOBAL_OFFSET_TABLE_[-1]`	This entry shall hold a `blrl` instruction. To obtains the address of the Global Offset Table, a function can call this entry (`bl _GLOBAL_OFFSET_TABLE_-4`) and have the address of the symbol `_GLOBAL_OFFSET_TABLE_` returned in `LR`.
`_GLOBAL_OFFSET_TABLE_[0]`	This entry shall hold the address of the Dynamic Segment, referenced by the symbol `_DYNAMIC`. This entry is initialized by the static linker.
`_GLOBAL_OFFSET_TABLE_[1]`	Reserved and shall be set to zero.
`_GLOBAL_OFFSET_TABLE_[2]`	Reserved and shall be set to zero.

A load module’s Global Offset Table must reside at a fixed relative location to the load module’s code. The code establishes addressability to the GOT by performing a branch and link to `_GLOBAL_OFFSET_TABLE_-4` which returns the address of `_GLOBAL_OFFSET_TABLE_` in `LR`. Because the branch and link instruction must be constructed by the static linker, a fixed relative relationship exists between the code and Global Offset Table. This can be satisfied by the static linker placing the GOT in the same segment as the code, a read-only segment, or by placing the GOT in some other segment

and having the system loader respect the relative placement of segments as specified in the load module by the static linker.

Note: The use of the branch and link instruction to establish addressability to the GOT requires that the label `_GLOBAL_OFFSET_TABLE_` is within ±32MB of the `bl _GLOBAL_OFFSET_TABLE_-4` instruction in all functions accessing the GOT.

Reviewer's Note: *Alternate schemes of establishing addressability to the GOT have been proposed. In essence, instead of* `bl _GLOBAL_OFFSET_TABLE_-4` *to establish addressability to the GOT, a* `bl` *to a find_got routine that is at a fixed relative location to the code is used. The find_got routine is patched by the loader to return the address of the GOT, which could now be placed anywhere (most interesting is shared, coerced memory). This scheme could be used in addition to the current scheme. See the example below.*

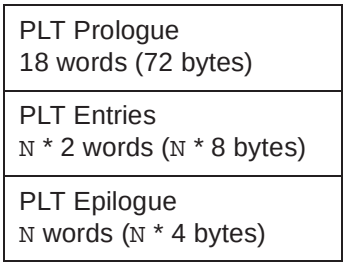
Description	Code Sequence
Current ABI	`  bl  _GLOBAL_OFFSET_TABLE_-4` `...` `  blrl` `_GLOBAL_OFFSET_TABLE_:` `...` `  mflr  %r31`
Additional technique for establishing GOT addressability. The compiler would call an alternate function to return the address in `r12`. This function would be patched by the loader to return the correct address of the GOT.	`  bl  _GLOBAL_OFFSET_TABLE_ADDRESS` `...` `_GLOBAL_OFFSET_TABLE_ADDRESS:` `  addis %r12, 0, _GLOBAL_OFFSET_TABLE_@ha` `  addi  %r12, %r12, _GLOBAL_OFFSET_TABLE_@l` `  blr` `...` `  mr  %r31, %r12`

11.6 Procedure Linkage Table (PLT)

The Procedure Linkage Table redirects function calls to functions whose location is unknown by the static linker. The static linker cannot resolve execution transfer between load modules and uses the PLT as an intermediary. The static linker arranges for control to be transferred to a PLT entry which will be modified by the dynamic linker to transfer control to the proper destination. This preserves the position independence of the code and allows load modules to reside at arbitrary memory locations. Each load module has its own Procedure Linkage Table to direct calls to functions external to the load module. The Procedure Linkage Table resides in the `.plt` section referenced by the `DT_PPC_PLT` entry in the Dynamic Segment.

For this ABI, the Procedure Linkage Table is not initialized in the load module by the static linker. Instead, the static linker simply reserves space for the PLT in the load module and the dynamic linker initializes and manages its memory image. The exact contents of the PLT are implementation dependent subject to the following rules.

1. The Procedure Linkage Table is partitioned in the following manner with `N` being the number of PLT entries in the load module.



2. The PLT Prologue is reserved for use by the dynamic linker. There shall be no branches from the load module into the Prologue.
3. For each function call in the load module redirected through the PLT, there shall be one PLT entry, 2 words in length. The static linker shall direct all branches for redirected function calls to the first word of its corresponding PLT entry, *e.g.* for PLT entry `i`, `i` ranging from 1 to `N` inclusive, the branch would be directed to `.plt` + 72 + (`i` - 1) * 8.

 If `N` > 8192, then for PLT entry indices `i`, where `i` > 8192, only even indices will be used and the corresponding PLT entries shall be 4 words in length.
4. The PLT Epilogue is reserved by use by the dynamic linker. There shall be no branches from the load module into the Epilogue.

Note: The use of branch and link instructions to transfer control to PLT entries requires that the PLT entries are within ±32MB of the `bl` instructions.

The relocation entries located via the `DT_JMPREL` tag in the Dynamic segment provide the information necessary for the dynamic linker to setup the Procedure Linkage Table. There is a one-to-one correspondence between the PLT entries and the `DT_JMPREL` relocation entries, *i.e.* the first relocation entry provides the relocation information for the first PLT entry and so on. Each relocation entry is of type `R_PPC_JMP_SLOT` and the `r_offset` member holds the location of the first byte of the associated PLT entry, *e.g.* `.plt` + 72 + (`i` - 1) * 8.

Following is an example used to illustrate how the Procedure Linkage Table might be utilized by a dynamic linker. This is an example only and does not specify the contents of the PLT or the behavior of the dynamic linker. Figure 11-4, “Procedure Linkage Table Example”, shows how the dynamic linker might initialize the PLT for an OS/2 load module.

Figure 11-4: Procedure Linkage Table Example

```
# PLT Prologue (18 words)
.PLTcall:
      addis %r11, %r11, .PLTtable@ha
      lwz   %r11, .PLTtable@l(%r11)
      mtctr %r11
      bctr
      nop
      nop
      nop
      nop
      nop
      nop
      nop
      nop
      nop
      nop
      nop
      nop
      nop

# PLT Entries (2 * N words)
.PLT1:
      addi  %r11, 0, 4*0
      b     .PLTcall
...

.PLTj:
      addi  %r11, 0, 4*(j - 1)
      b     .PLTcall
...

.PLTk:
      b     <target>
      nop
...

.PLTN:
      addi  %r11, 0, 4*(N - 1)
      b     .PLTcall

# PLT Epilogue (N words)
.PLTtable:
      <N word table>
```

The PLT is initialized by the dynamic linker during process initialization.

1. The dynamic linker writes the instructions for `.PLTcall` in the PLT Prologue adjusting them for the virtual address of `.PLTtable`.
2. The dynamic linker locates the PLT relocation entry for each PLT entry `i`. The relocation entry will be of type `R_PPC_JUMP_SLOT`, its `r_offset` member will contain the address of `.PLTi` and its symbol table index will point to the target symbol, *e.g.* `foo`.
3. The dynamic linker determines the proper value of symbol `foo` and initializes the code at `.PLTi` in one of two ways.
 - If `foo` is reachable from `.PLTi` by a single branch instruction, a branch to `foo` instruction is placed in the first word at `.PLTi`.
 - If `foo` is not reachable from `.PLTi` by a single branch instruction, the address of `foo` is stored in the PLT Epilogue at `.PLTtable` + (`i` - 1) * 4, an instruction which loads `r11` with (`i` - 1) * 4 is placed in the first word at `.PLTi` and a branch to `.PLTcall` instruction is placed in the second word at `.PLTi`.

 If `i` > 8192, then `i` must be even and the size of `.PLTi` is 4 words. More than one instruction will be necessary to load `r11` with (`i` - 1) * 4.
4. The PLT entry is now initialized to transfer control to the target function either directly, via a direct branch, or indirectly through `.PLTcall`.

Note: Lazy binding of PLT entries is not supported by this ABI.

Appendix A Compiler Support Extensions

This appendix contains ABI extensions added to support additional compiler function.

This appendix also includes the description of the recently adopted COMDAT extension to ELF.

A.1 ELF

A.1.1 Sections

The following section types are defined. These sections are used to communicate information from the compiler to the static linker. These sections are only valid in object files.

Table A-1: Section Types, `sh_type`

Name	Value	Meaning
`SHT_COMDAT`	12	This section contains COMDAT code or data and is otherwise treated the same as `SHT_PROGBITS`. The static linker will choose only one `SHT_COMDAT` section of a given name from all like named `SHT_COMDAT` sections encountered during the link.
`SHT_IDMDLL`	`0x60001002`	This section holds the information to perform symbol name demangling. See § A.1.1.2, “Symbol Name Demangling”, on page 144 for details.
`SHT_DEFLIB`	`0x60001003`	This section holds the information about static libraries to be processed by the static linker. See § A.1.1.3, “Default Library”, on page 145 for details.

Two members in the section header, `sh_link` and `sh_info`, hold special information depending on the section type.

Table A-2: `sh_link` **and** `sh_info` **Interpretation**

`sh_type`	`sh_link`	`sh_info`
`SHT_COMDAT`	The section header index of the section into which this section shall be combined in the output file. This will allow many small sections to be combined into one. The value `SHN_UNDEF` indicates that this section is not to be combined into another section.	COMDAT selection criteria. See Table A-3, “COMDAT Selection Criteria” below for possible values.

Table A-2: `sh_link` and `sh_info` Interpretation (Continued)

sh_type	sh_link	sh_info
SHT_PROGBITS	The section header index of the section into which this section shall be combined in the output file. This will allow many small sections to be combined into one. The value SHN_UNDEF indicates that this section is not to be combined into another section.	0
SHT_IDMDLL SHT_DEFLIB	The section header index of the string table used by entries in this section.	0

The SHT_IDMDLL and SHT_DEFLIB sections must be word aligned. Thus the sh_addralign value for these sections is 4.

A.1.1.1 COMDAT Section

Object files may contain one or more SHT_COMDAT sections uniquely named within the object file. These section exist to support C++ templates and other like constructs. A compiler can emit the same SHT_COMDAT section into several object files using the same section name. The static linker will select one of the like named sections to be placed in the resulting load module. The selection is based upon the following criteria specified in the sh_info field of the section header table entry of the COMDAT section.

Table A-3: COMDAT Selection Criteria

Name	Value	Meaning
COMDAT_NONE	0	Invalid selection criteria.
COMDAT_NOMATCH	1	Only one instance of a SHT_COMDAT section of the given name is allowed. A link warning is generated otherwise.
COMDAT_PICKANY	2	Pick any instance of a SHT_COMDAT section of the given name.
COMDAT_SAMESIZE	3	Pick any instance of a SHT_COMDAT section of the given name but all instances of SHT_COMDAT sections of the given name must have the same size. A link warning is generated otherwise.

The following symbol bindings are also defined.

Table A-4: Symbol Binding, `ELF32_ST_BIND`

Name	Value	Meaning
`STB_GLOBALOMIT`	3	A global symbol that may be omitted from the output file. If the symbol is undefined and there are no references to the symbol (no relocation entries specifying its index), then the symbol need not be defined (nor is it searched for in libraries). It is otherwise treated the same as `STB_GLOBAL`.

A.1.1.2 Symbol Name Demangling

Each object file may contain one `SHT_IDMDLL` section. The section contains exactly one `Elf32_Demangle` structure.

Figure A-1: Demangle Information Structure, `Elf32_Demangle`

```
typedef struct
{
   Elf32_Word     idm_dllname;
   Elf32_Word     idm_initparms;
} Elf32_Demangle;
```

Member	Description
`idm_dllname`	This member identifies the string table index of the name of the dynamic link library containing the symbol name demangling function. The name string is UTF-8 encoded.
`idm_initparms`	This member holds the string table index of the parameters to the dynamic link library initialization function.

The very first section of this type encountered by the linker will define the demangle dynamic link library used by the static linker. The demangle dynamic link library contains two entry points. The static linker shall call `InitDemangleID32` before using the `DemangleID32` function to demangle symbol names.

Table A-5: `InitDemangleID32` Function Prototype

```
unsigned long InitDemangleID32 ( unsigned char *pszInitParms );
```

Parameter	Description
`pszInitParms`	This parameter is the null-terminated string provided by the `idm_initparms` member of the `Elf32_Demangle` structure.

Table A-5: `InitDemangleID32` Function Prototype (Continued)

Return Value	A non-zero return value indicates successful initialization. A zero return value indicates that initialization failed. If a failure is indicated, the `DemangleID32` function in the dynamic link library should not be called.

Table A-6: `DemangleID32` Function Prototype

```
unsigned long DemangleID32 ( unsigned char *pszMangledName,
                             unsigned char *pbDemangledName,
                             unsigned long ulDemangledName );
```

Parameter	Description
`pszMangledName`	This parameter is the null-terminated symbol name to be demangled.
`pbDemangledName`	This parameter is a pointer to a buffer where the null-terminated demangled name is to be returned.
`ulDemangledName`	This parameter is the size of the buffer.
Return Value	A non-zero return value indicates successful initialization. A zero return value indicates that initialization failed. If a failure is indicated, the `pbDemangledName` buffer contents are invalid.

A.1.1.3 Default Library

Each object file may contain one `SHT_DEFLIB` section. The section contains an array of `Elf32_Library` structures. Each structure represents a library to be added to the end of the list of default libraries to be searched by the static linker to resolve undefined symbols.

Figure A-2: Default Library Structure, `Elf32_Library`

```
typedef struct
{
   Elf32_Word     lib_name;
} Elf32_Library;
```

Member	Description
`lib_name`	This member identifies the string table index of the name of a static library. The name string is UTF-8 encoded.

A.1.2 Note Information

The following note information is defined. This information, if present, resides in a `PT_NOTE` segment of a load module.

A.1.2.1 Browser Information

A list of the full file names of the object files that contributed to the load module is kept in the browse information. This information can be used by a code browser.

Table A-7: Browser Information

Field	Description
`namesz`	The size of `name` including the null-terminating byte.
`descsz`	The size of `desc` information.
`type`	1
`name`	The null-terminated string "`IBM`".
`desc`	This information is a collection of records described in § A.1.2.1.1, "Browser Information Records".

A.1.2.1.1 Browser Information Records

There are three record types. Each record begins with a single byte containing the record type. It is immediately followed by a null-terminated string containing a full file name. The full file name strings are UTF-8 encoded.

Table A-8: Browser Information Record Types

Type	Description
`0`	The following string is the full file name of an object file included in the load module.
`1`	The following string is the full file name of an object library file which has *some* of its member object files included in the load module. This record type is followed by additional information describing which member object files are included in the load module. If necessary, a padding byte of zero follows the string to align the additional information on a halfword boundary. The additional information begins with a halfword containing the number, `m`, of member object files included in the load module. This is followed by `m` halfwords each containing the member number of the member object file. (*e.g.* 1 represents the first member object file in the object file library.)
`2`	The following string is the full file name of an object library file which has *all* of its member object files included in the load module.

A.1.2.2 Version Information

File version information may need to be included in a load module. This information can be used for file version identification and product service.

Table A-9: Version Information

Field	Description
`namesz`	The size of `name` including the null-terminating byte.
`descsz`	The size of `desc` including the null-terminating byte.
`type`	2
`name`	The null-terminated string "`IBM`".
`desc`	A null-terminated, UTF-8 encoded version string.

A.1.2.3 Description Information

The static linker may have a descriptive comment from the user to be included in a load module. This information can be a description of the load module.

Table A-10: Description Information

Field	Description
`namesz`	The size of `name` including the null-terminating byte.
`descsz`	The size of `desc` including the null-terminating byte.
`type`	3
`name`	The null-terminated string "`IBM`".
`desc`	A null-terminated, UTF-8 encoded description string.

End of Document

This is the last page of the document.